TWO-TIMESCALE ADAPTIVE DECODING FOR QUANTUM ERROR CORRECTION WITH SURFACE CODES

Anonymous authors
Paper under double-blind review

ABSTRACT

A real-time Quantum Error Correction (QEC) decoder should adapt to complex and evolving hardware noise while remaining sufficiently lightweight to meet stringent latency requirements. To decouple adaptivity from latency while enabling label-free runtime adaptation, we propose *SF-QEC*, a slow-fast two-timescale decoding framework. SF-QEC employs two structurally aligned recurrent-transformer models inspired by AlphaQubit 2: a neural decoder operates on the fast path for streaming decoding, while an adaptation model operates asynchronously on the slow path to analyze longer histories of detection events. The slow model extracts history-derived context and transfers it to the corresponding recurrent states of the fast decoder, augmenting its temporal representation without requiring runtime labels or placing the slow model on the latency-critical path. On QEC data collected from the IBM Phoenix superconducting quantum processor, the full SF-QEC framework further improves already-superior decoding performance compared to baselines, demonstrating the potential of two-timescale neural decoding under evolving hardware conditions.

1 INTRODUCTION

Quantum computers are highly sensitive to noise: even small errors in physical qubits can accumulate and corrupt a computation long before it finishes. Quantum Error Correction (QEC) (Terhal, 2015) protects quantum information through an encoding-checking-decoding process. Before computation begins, the logical information is redundantly encoded across multiple physical qubits. During computation, subsets of these qubits are repeatedly measured, producing a stream of error-detection events. A decoder then interprets this event stream to infer the most likely errors and determine the corrections needed to preserve the logical information. Because these errors are not directly accessible and must be inferred from the detection event data, the reliability of the encoded computation depends critically on the decoder's ability to make accurate corrections.

Analytical decoders such as Minimum-Weight Perfect Matching (MWPM) (Dennis et al., 2001) are effective when the hardware noise is well-characterized. Neural decoders offer a complementary advantage: through data-driven training, they can learn complex correlations and latent structures that are difficult to model explicitly (Meinerz et al., 2022; Bausch et al., 2024). However, most existing QEC decoders are trained or calibrated offline and then held fixed during deployment, without explicitly adapting their noise assumptions as hardware conditions evolve. Such changes can be commonly observed – for example, Figure 1 (detailed in Appendix A) shows substantial temporal variation in the Logical Error Rate (LER, the fraction of shots decoding fails to correct) over one hour on IBM Phoenix quantum computer (IBM Quantum, 2026). Although calibration can update hardware configuration and error model, it largely remains an offline approach, requiring dedicated QPU time and providing only periodic snapshots of device con-

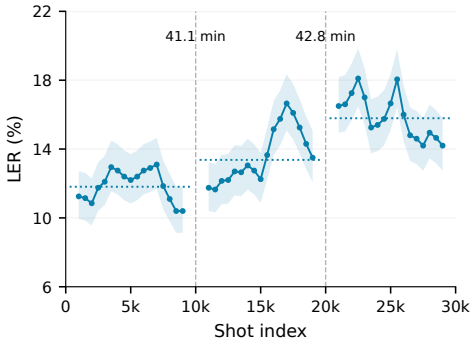


Figure 1: Shifting of logical error rate in 1.5 hours on IBM Phoenix quantum computer.

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

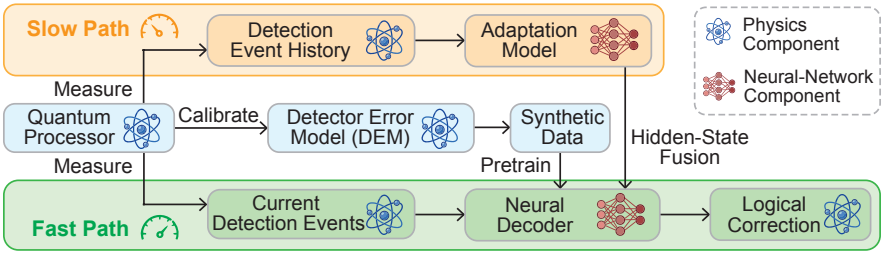


Figure 2: Overview of SF-QEC. The slow path model analyzes error detection history, asynchronously augmenting hidden states in the decoder on the fast path at runtime.

ditions, potentially missing transient changes during operation. Capturing hardware evolving conditions during operation is therefore a natural way – however, two challenges need to be addressed.

Latency becomes a tight constraint for runtime adaptation. In superconducting processors, QEC cycles occur on a timescale of approximately 1 μs (Krinner et al., 2021), requiring the decoder to keep pace with a rapidly arriving stream of detection events. Recent work has demonstrated low-latency neural decoding on specialized accelerators, including FPGAs and TPUs (Overwater et al., 2022; Zhang et al., 2023; Senior et al., 2025). Meanwhile, emerging heterogeneous interconnects make it increasingly feasible to separate latency-critical inference from more computationally intensive background processing. For example, recent NVQLink results report GPU-FPGA loopback latencies below $4\,\mu$s (Caldwell et al., 2025), suggesting that contextual information generated by a more powerful processor can be exchanged with a real-time decoding device on a μs timescale.

Labels are unavailable during practical execution. During a quantum algorithm, the physical errors cannot be observed directly, so the system does not know with certainty which qubits experienced errors or what the correct recovery should have been for each shot. Such labeled supervision is typically available only in controlled experiments, such as quantum-memory experiments, where the prepared logical state and final logical outcome are known and can be used to determine whether decoding succeeded. As a result, a deployed decoder cannot rely on new labeled data during operation for online retraining, making runtime adaptation substantially more challenging.

Motivated by these two constraints, we propose *SF-QEC*, a two-timescale decoding framework consisting of a slow path and a fast path. We use surface-code QEC as a representative case study due to its central role in fault-tolerant quantum computing. As illustrated in Figure 2, a neural decoder operates on the fast path to process individual QEC cycles, while an adaptation model operates asynchronously on the slow path to analyze longer histories of detection events and provide contextual information to the fast decoder. A practical implementation could, for example, place the slow adaptation model on a GPU and the latency-critical decoder on a specialized accelerator such as an FPGA. In this work, however, we focus on the algorithmic design and evaluation of the two-timescale framework rather than a fully integrated hardware implementation.

In this work, both slow and fast paths are instantiated with recurrent-transformer architectures inspired by AlphaQubit 2 (Senior et al., 2025), which combines recurrent layers for temporal information with transformer layers for spatial interaction. Because the two models share aligned internal representations, the slow-path context (Section 3.2) can be exported directly to the corresponding recurrent states of the fast decoder through state-fusion (Section 3.3), allowing the neural decoder to benefit from longer-term information without placing the slow model on the latency-critical path.

Our main contributions are listed below.

- We propose SF-QEC, a two-timescale decoding framework that decouples adaptivity from latency by deploying two models on separate slow-fast paths, where a neural decoder on the fast path is augmented by an adaptation model on the slow path.
- We design a slow-to-fast augmentation mechanism. Deployed on Real-time AlphaQubit 2 architecture, the slow model captures history context through hierarchical history embedding, and transfers the context to the fast decoder through state fusion, without requiring runtime labels.
- We evaluate SF-QEC framework on experimental surface-code data obtained on IBM Phoenix superconducting quantum computer, demonstrating further improved decoding accuracy compared to standalone Real-time AlphaQubit 2 under nonstationary noise.

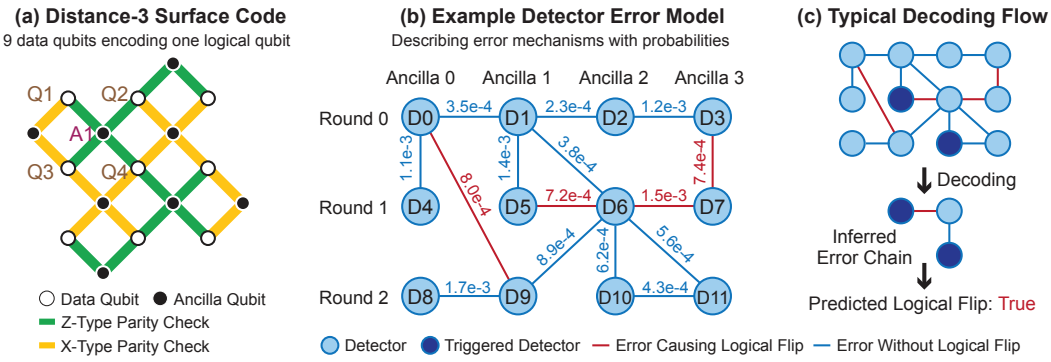


Figure 3: (a) Surface code example. (b) DEM example. (c) Typical decoding flow with the DEM and error detection events. The measurement of an ancilla qubit effectively checks the parity of neighboring data qubits. A detector is triggered if the associated ancilla qubit measurement changes.

2 Background

2.1 Quantum Error Correction with Surface Code

Surface code (Bravyi & Kitaev, 1998) is a well-established QEC scheme in which physical qubits are arranged on a two-dimensional lattice. Figure 3(a) illustrates a distance-3 surface code encoding one logical qubit using nine data qubits. Errors are detected through repeated measurements of local *stabilizers*, i.e., parity-check operators acting on neighboring data qubits. Each stabilizer is measured using an associated ancilla qubit; for example, A1 measures a Z-type stabilizer over Q1-Q4 and is therefore sensitive to X errors on these data qubits (similarly, an X-type stabilizer is sensitive to Z errors). A change in a stabilizer measurement between consecutive QEC rounds produces a detection event used by the decoder. Increasing the code distance generally improves error protection at the cost of additional physical qubits.

Detector Error Models (Gidney, 2021) summarize the relationship between physical errors and detection events. As Figure 3(b) shows, a DEM specifies possible error mechanisms (edges), their probabilities, whether they can cause logical flipping, and detectors they trigger. A DEM can also have boundary edges triggering one detector, and hyperedges triggering more than two detectors (not shown for simplicity). For each round of ancilla qubit measurement, i.e., parity check, a detector associated with an ancilla qubit checks whether the measured parity changes between consecutive rounds. As a comprehensive prior, a DEM is typically used by a decoder to infer the most likely errors without directly measuring the encoded logical state.

QEC Decoders take the observed detection events and infer how the underlying physical errors may have affected the logical information. Analytical decoders typically construct this inference procedure explicitly from a DEM. MWPM (Dennis et al., 2001), for example, assigns weights to possible error mechanisms according to their estimated probabilities. Given a set of triggered detectors, MWPM finds a minimum-weight error chain consistent with the observations and uses the inferred chain to determine the corresponding logical correction. Figure 3(c) illustrates an example with two triggered detectors. Based on the DEM, the decoder identifies the most likely error chain connecting them. Because the inferred chain contains an odd number of mechanisms associated with a logical flip (shown in red), the decoder predicts a logical error.

2.2 Real-Time AlphaQubit 2

AlphaQubit 2 (Senior et al., 2025) is a neural QEC decoder designed to process a sequence of error-detection measurements and predict logical observables. Its architecture maintains an internal representation for each *stabilizer*, where a stabilizer is a local parity check over a group of data qubits, whose outcome is measured through an associated ancilla qubit during each QEC cycle. AlphaQubit 2 separates temporal and spatial information processing by interleaving recurrent layers with transformer layers. The recurrent layers propagate information from earlier QEC cycles (aka rounds) through a per-stabilizer state, while the transformer layers perform spatial mixing among stabilizer representations within the same time step.

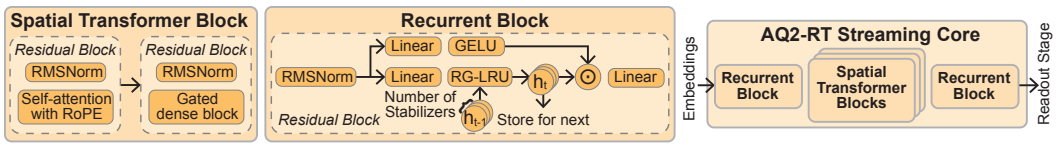


Figure 4: Major components of Real-time AlphaQubit 2 architecture. RoPE: Rotary Position Embedding; GELU: Gaussian Error Linear Unit; RG-LRU: Real-Gated Linear Recurrent Unit.

Real-Time AlphaQubit 2 (AQ2-RT) is a speed-optimized variant designed for real-time decoding (see Figure 4). It retains the same overall separation between recurrent temporal processing and transformer-based spatial processing, but substantially reduces the computational cost of the network. In particular, AQ2-RT utilizes a lightweight gated recurrence based primarily on element-wise operations. It also uses a shallower network and smaller internal representations.

The recurrent state is particularly relevant to our work because it provides an explicit representation h_t through which information from previous QEC cycles influences future decoding. Here, h_t is the recurrent state for recurrent step t, which is updated constantly at runtime. SF-QEC builds on this interface: rather than relying only on the state accumulated by the real-time decoder itself, a slow-path adaptation model analyzes a longer detection history and exports additional contextual information to augment the corresponding recurrent states of the fast decoder.

3 METHODS

In this section, we provide an overview of the SF-QEC workflow. Then, we describe the modifications applied on the original AQ2-RT, on both the slow path and fast path.

3.1 SF-QEC OVERVIEW

Figure 5 presents the overall workflow of SF-QEC. The framework combines a fast-path neural decoder with a slow-path history encoder for adaptation, both built from structurally aligned recurrent-transformer architectures inspired by AlphaQubit 2. The fast path processes newly arriving detection events at the QEC-cycle timescale, while the slow path analyzes a longer detection history using the embedding mechanism (Section 3.2). Their corresponding recurrent blocks provide a natural interface for transferring history-derived context from the slow path to the fast decoder.

On the *slow path*, detection histories from multiple shots are converted into compressed history embeddings and processed by recurrent and spatial transformer blocks. The recurrent blocks capture temporal information across the history window, while the transformer blocks model spatial correlations among stabilizer representations. After the complete window is processed, the final state from each recurrent block is retained as a *slow state*, cached, and asynchronously used by the fast decoder. Because this computation lies outside the latency-critical decoding loop, the slow path can employ an adaptation model that processes more information. Moreover, it relies only on observed detection histories and therefore requires no logical labels at runtime.

On the *fast path*, unlike the shot-level embeddings used on the slow path, round-level embeddings are processed. At each recurrent block, the updated fast state is concatenated with the corresponding cached slow state and passed through a lightweight linear projection, as illustrated at the bottom of Figure 5. The resulting fused state (detailed in Section 3.3) is then carried forward as the recurrent state for subsequent QEC cycles. In this way, longer-term context can influence real-time decoding without modifying the fast decoder's parameters or requiring online retraining.

3.2 HIERARCHICAL HISTORY EMBEDDING

Figure 6 illustrates the embedding mechanisms used in SF-QEC. The fast path follows the AQ2-RT embedding design and is shown only for reference. As illustrated on the left of the red dashed line, detection events from each QEC round are mapped to per-stabilizer embeddings, while the final-boundary events are embedded using a separate set of parameters. The slow path uses the same raw embedding functions, so this shared step is omitted from the lower part of the figure.

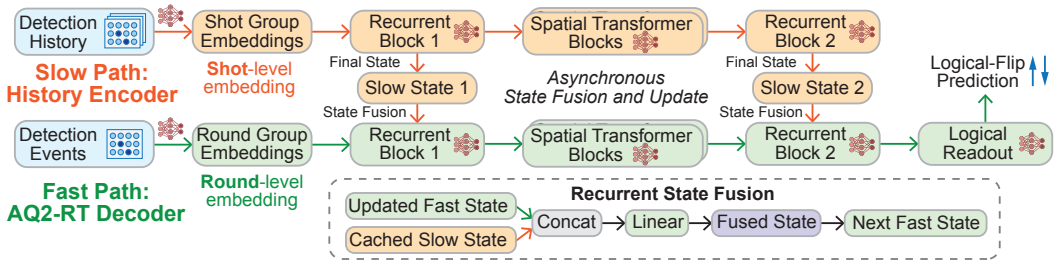


Figure 5: Workflow of SF-QEC. The slow states from recurrent blocks are final states of a slow path run over a history window, encoding shot-level temporal context.

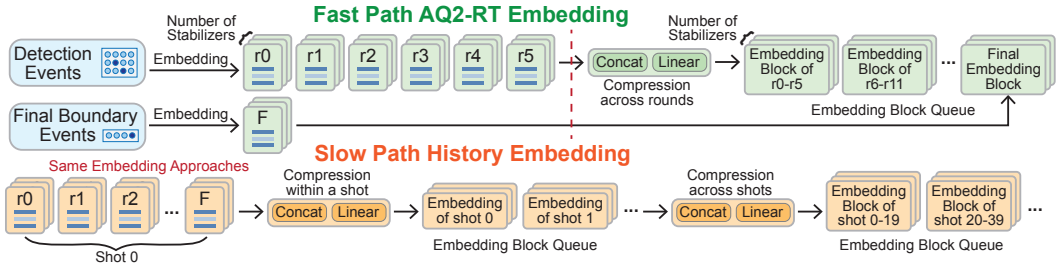


Figure 6: Embedding mechanisms in SF-QEC. The fast path follows the AQ2-RT procedure. The slow path introduces hierarchical temporal compression, aggregating rounds within each shot into shot-level embeddings and then compressing groups of shots into history embedding blocks.

Following AQ2-RT, fast-path embeddings from several consecutive QEC rounds are concatenated and linearly projected into a compressed embedding block. Repeating this operation produces the sequence of embedding blocks consumed by the fast recurrent-transformer decoder, with the final-boundary embedding incorporated into the last block. The figure uses 6-round blocks, for example.

The slow path embedding counterparts, however, must represent a much longer history spanning multiple completed QEC shots. To implement, we adopt a hierarchical temporal embedding scheme. We first perform *within-shot compression*: the round-level embeddings belonging to one shot, together with its final-boundary embedding, are concatenated and projected into a single shot-level representation. This converts the detection history into a sequence of compact shot embeddings.

We then apply *across-shot compression*. Consecutive shot-level embeddings are grouped, concatenated, and projected again to form history embedding blocks (Figure 6 uses 20-shot blocks, for example). These blocks constitute the input sequence to the slow-path recurrent-transformer model. In this way, the slow path preserves the same per-stabilizer representation used by the fast decoder while extending its temporal scope from individual rounds to many shots. The hierarchical compression therefore allows the slow model to capture longer-term temporal context without processing every historical QEC round individually.

3.3 Cross-Path Recurrent State Fusion

Figure 7 illustrates how SF-QEC transfers information from the slow path into the recurrent state of the fast decoder. Compared to the recurrent block in Figure 4, slow states are incorporated in the data path through a lightweight fusion operation. Because the slow and fast models maintain aligned recurrent-state representations, each cached slow state corresponds directly to the recurrent state of the same stabilizer and recurrent block, enabling direct state correspondence without additional matching or attention. The slow states are generated asynchronously from a longer detection-history window and cached until the next slow-path update becomes available. During this interval, the fast decoder continues to operate normally using the latest cached context, with every update involving the slow states that carry historical context, thus preserving history-awareness.

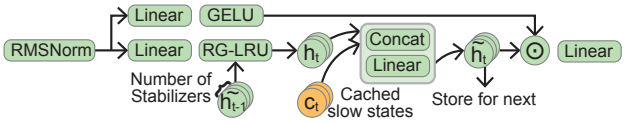


Figure 7: Recurrent state fusion in SF-QEC.

4 RESULTS

4.1 EXPERIMENTAL SETUP

Data acquisition. As a case study, we adopt distance-3 logical-Z memory experiments on IBM Phoenix, a 120-qubit superconducting quantum processor. Each circuit measures four disjoint patches of 9 data qubits and 8 ancilla qubits, with $r \in \{3, 6, 9\}$ detection rounds and no intermediate ancilla resets. Although surface-code memory experiments commonly use $r =$ distance, we additionally evaluate $r = 6$ and $r = 9$ to study decoding robustness over longer memory durations. Each setting contains 20 batches with 10K shots each, where batch-16 is discarded due to abnormal error rates (Appendix B documents batch-16's exclusion). Each shot takes approximately 50 μs to finish, including overhead. Each new batch is acquired approximately 40 minutes after the latest batch to allow hardware to evolve. Separate patches have separate data, noise priors (DEM), and are processed with different SF-QEC models. For synthetic pretraining, Stim (Gidney, 2021) samples 2.1M fresh shots per epoch from each patch and each detection round setting's fixed batch-11 DEM, with 100K validation and 1M test shots.

Training and evaluation protocol. Training comprises synthetic fast-decoder pretraining, supervised fast-model fine-tuning on batches 1-11, and joint training on the same batches. Fast fine-tuning uses their first 8,000 shots (88,000 total); the last 2,000 per batch provide validation. Joint training uses 500 preceding shots with per 20-shot target block. Each stage has a 50-epoch cap and 5-epoch patience, selecting by validation binary cross-entropy. All weights remain fixed at test time, and recurrent state fusion uses no logical labels. All methods score the same 64,000 targets: the last 8,000 shots of batches 12-15 and 17-20. At runtime, slow states are updated every 20 shots.

Model configurations and environments. We use fast/slow model embedding widths 64/64 (SF-QEC-64) and 128/128 (SF-QEC-128) in the main comparison. Training uses A100 and H100 GPUs; optimized inference is measured on A100 and GH200 using the same protocol. We apply PyTorch Inductor compilation for fixed input shapes and frozen weights. We also adopt CUDA graph replay to reduce repeated kernel-launch overhead.

We compare SF-QEC against the following representative QEC decoders:

- **MWPM** (Dennis et al., 2001): a standard minimum-weight perfect matching decoder.
- **Correlated Matching** (Fowler, 2013; Higgott & Gidney, 2025): a two-pass matching method that uses the first decoding pass to reweight correlated error mechanisms before a second pass.
- **Union-Find** (Delfosse & Nickerson, 2021): an almost-linear-time decoder that grows and merges clusters of detection events before constructing a correction.
- **BP-OSD / BP-LSD** (Roffe et al., 2020a; Hillmann et al., 2025a): belief-propagation-based decoders that use statistics-based post-processing to resolve cases where BP alone is insufficient, with LSD providing a more localized alternative to OSD.
- **DGR alignment / heuristic** (Wang et al., 2023): adaptive decoding-graph reweighting methods that update edge weights using statistics from previous decoding results, with the heuristic variant additionally applying correlation-aware reweighting and a subsequent matching pass.

4.2 LOGICAL ACCURACY

Overall accuracy comparison. Table 1 compares four SF-QEC settings and seven baseline algorithms. The same fixed prior (Batch-11 DEM) and corrected detector coordinates are used across the methods. All SF-QEC configurations outperform every listed baseline for all experiments with a notable gap, mainly attributing to the powerful AQ2-RT. FP16 inference reuses the FP32-trained checkpoints without retraining, with a maximum LER change of only 0.014.

SF-QEC effectiveness. Since SF-QEC aims to adapt the decoder to evolving hardware condition, we compare the joint SF-QEC model with AQ2-RT on the fast path in Figure 8, with AQ2-RT LER shown alongside to indicate the baseline decoding difficulty. SF-QEC-64 improves pooled LER in 11 settings and ties in one, with mean absolute LER reduction being 0.236. It can be observed that SF-QEC tends to improve batches with higher LER with AQ2-RT alone – notably, over 1.0 LER is reduced for batch 15 in the r = 3, patch-0 experiment. Figure 9 further shows the comparison for

Table 1: Pooled experimental LER (%; lower is better) on 64,000 shots per setting (batch 12-15 and batch 17-20). Bold marks the lowest LER in each column. Four configurations of SF-QEC are utilized. P0-P3: patch-0 to patch-3; r=3/6/9: 3/6/9 detection rounds per shot.

	$r = 3$				$r = 6$				$r = 9$			
Decoder	P0	P1	P2	P3	P0	P1	P2	P3	P0	P1	P2	P3
MWPM	12.373	4.130	8.869	8.820	22.011	9.459	18.183	16.481	29.666	14.872	25.433	22.344
Correlated Matching	12.762	4.030	8.502	8.941	22.509	9.114	18.408	16.528	29.873	14.180	25.688	21.925
Union-find	14.395	4.659	10.592	9.705	24.730	10.602	21.322	17.294	32.028	16.197	29.209	23.292
BP–OSD	12.620	3.930	8.230	8.748	21.934	8.623	16.819	16.170	29.398	13.252	23.566	21.497
BP–LSD	12.764	3.931	8.231	9.037	22.142	8.611	16.800	16.242	29.609	13.264	23.580	21.483
DGR alignment	12.048	3.959	7.906	8.750	21.692	8.616	17.100	16.353	29.211	13.625	24.159	21.850
DGR heuristic	12.491	4.053	8.044	8.831	23.702	9.297	17.781	17.392	31.992	15.044	26.753	26.148
SF-QEC-64 FP32	10.877	3.431	**6.450**	**7.627**	20.194	7.766	14.642	14.319	28.455	12.127	21.603	20.641
SF-QEC-64 FP16	10.873	3.431	6.452	**7.627**	20.195	7.769	14.642	14.317	28.462	12.128	21.617	20.647
SF-QEC-128 FP32	**10.727**	**3.380**	6.491	7.661	**20.020**	7.686	14.250	**14.273**	27.788	11.934	**20.891**	20.348
SF-QEC-128 FP16	**10.727**	3.381	6.489	7.661	**20.020**	**7.684**	**14.248**	14.277	**27.780**	**11.933**	**20.891**	**20.347**

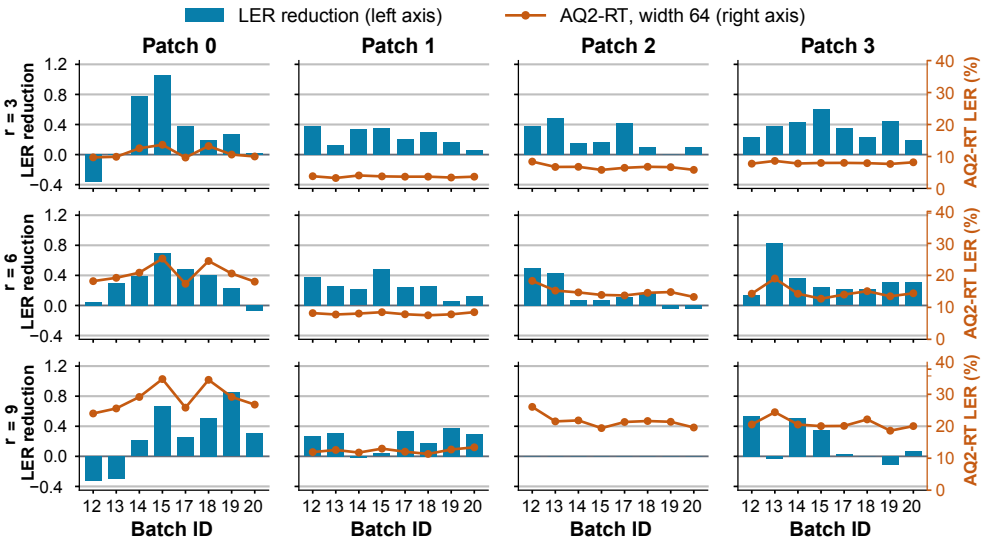


Figure 8: Per-batch LER reduction from AQ2-RT to SF-QEC, with embedding width = 64 on both fast and slow paths. Positive favors SF-QEC. Each bar scores 8,000 shots. Patch-2, r = 9 case shows a tie because joint training does not show better validation loss compared to AQ2-RT alone.

128 embedding width, with mean absolute LER reduction of 0.222, improving the fast path in most cases. Appendix C includes more width sweeps.

Figure 10 takes the r = 3, patch 0, 64-width experiment for example, demonstrating the LER situation within each batch. It can be observed that batch 14 and 15 achieve higher LER reduction, with visibly reduced overlap between the Wilson confidence bands.

4.3 Inference Costs

Table 2 separates complete-shot fast decoding with cached context from one 500-shot slow refresh. FP16 reuses FP32 models. Detailed model configurations are in Appendix B.4. Despite having fewer parameters compared to the fast model, more shot data flow through the slow model and incur more computation. On A100, FP16 lowers fast latency by 11.2–18.8% and slow latency by 14.2–23.8% across these configurations. On GH200, the corresponding reductions are 13.9–81.5% and 14.3–80.9%. The results demonstrate that meeting fast path latency requirement (e.g., 3 μs for $r = 3$) remains challenging on GPUs and the fast path should be deployed on devices such as TPUs, FPGAs, or ASICs. For the slow path, FP16 latency results are all within our 1 ms timing budget (updating once per 20 shots, 50 μs per shot), making GPUs practical slow-path platforms.

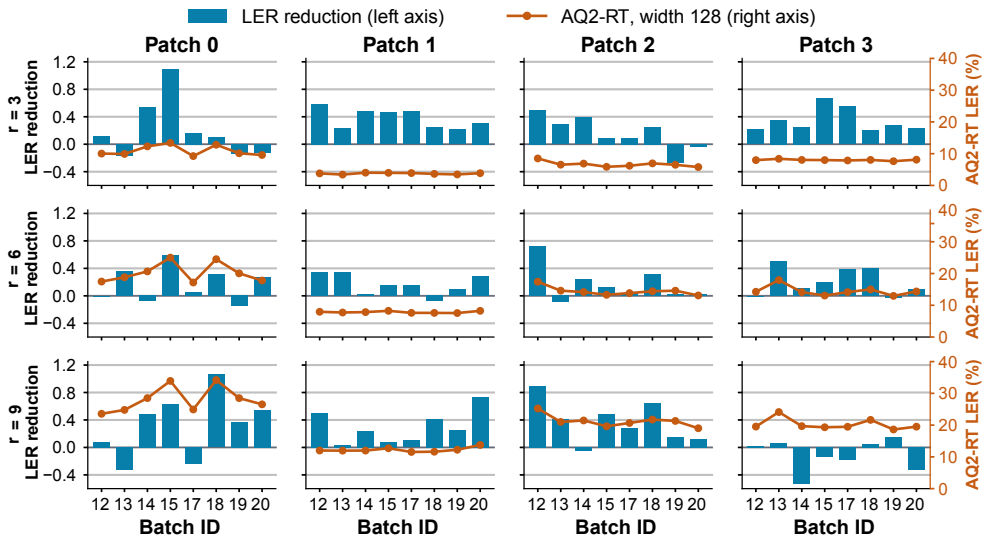


Figure 9: Per-batch LER reduction from AQ2-RT to SF-QEC, with embedding width = 128 on both fast and slow paths. Positive favors SF-QEC. Each bar scores 8,000 shots. Patch-3, r = 9 case is worse because validation loss decreases in joint training but failed to generalize.

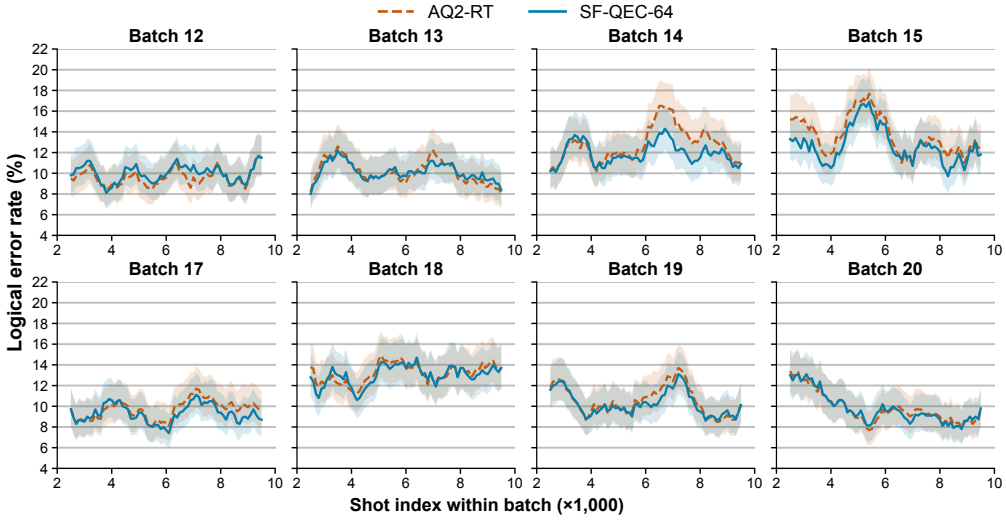


Figure 10: Rolling-window LER for the batches of r = 3, patch-0, embedding width = 64. Shaded regions are 95% Wilson confidence bands. Window size: 1000 shots; Stride: 100 shots.

4.4 Limitations and Future Work

This work focuses primarily on algorithmic validation of the proposed two-timescale QEC decoding framework. While we provide component-level inference measurements, a fully integrated deployment with the quantum-control stack remains future work.

Historical context improves decoding performance in the majority of our experimental settings, although degradations are observed in a few cases, indicating that not all history windows are equally informative. Future work can incorporate confidence-aware context selection or gating mechanisms that reduce the influence of uninformative historical context and fall back to the fast decoder.

Broader experimental scenarios are to be investigated. Our current evaluation focuses on distance-3 surface-code memory experiments on a single superconducting processor, providing a controlled setting to study the proposed slow-fast mechanism. Future work will extend the evaluation to larger code distances, additional quantum processors and noise regimes, and longer-duration experiments to further assess the scalability and generality of SF-QEC.

Table 2: Parameter counts and latency in μs. Width denotes the embedding width of both models; Fusion reports the added parameters, already included in Fast. Fast: one complete shot with cached context. Slow: one 500-shot refresh. Each latency cell gives FP32 / FP16.

		Parameters			A100-SXM4-80GB		GH200	
r	Width	Fast	Slow	Fusion	Fast	Slow	Fast	Slow
3	64	0.67 M	0.25 M	0.02 M	391.4 / 347.3	796.0 / 669.5	345.6 / 292.9	706.6 / 605.5
3	128	2.39 M	1.01 M	0.07 M	438.0 / 366.8	972.8 / 743.8	1545.7 / 306.0	3386.9 / 658.1
6	64	0.67 M	0.27 M	0.02 M	390.9 / 347.3	843.8 / 707.7	344.4 / 296.7	733.1 / 626.5
6	128	2.39 M	1.06 M	0.07 M	434.7 / 359.5	1037.8 / 794.5	1552.2 / 307.1	3422.1 / 652.4
9	64	0.67 M	0.28 M	0.02 M	498.9 / 432.3	871.4 / 747.9	435.1 / 366.2	793.9 / 626.3
9	128	2.39 M	1.10 M	0.07 M	566.0 / 459.3	1100.3 / 838.8	2102.7 / 388.4	3465.2 / 704.4

Continual slow-path adaptation. Through this work, We would like to advocate that a slow path should continually accumulate information as the quantum processor operates, building persistent knowledge of recurring and newly emerging noise patterns. This could enable co-evolution of the machine and model for an increasingly more robust QEC system.

5 Related Work

Analytical decoders typically rely on explicit noise descriptions and hand-designed inference procedures. Minimum-Weight Perfect Matching (Dennis et al., 2001) is the most widely used example, while Union-Find (Delfosse & Nickerson, 2021) provides an almost-linear-time alternative. Correlated matching (Paler & Fowler, 2023) incorporates dependencies between detector events, and Sparse Blossom (Higgott & Gidney, 2025) provides an efficient implementation of MWPM. More computationally intensive methods are often used for high-accuracy offline decoding: maximum-likelihood decoding (Bravyi et al., 2014) seeks the most probable logical correction by accounting for all error configurations consistent with the observed detection events. More methods include belief-matching (Higgott et al., 2023), which combines belief propagation with matching to incorporate higher-order error mechanisms, and Harmony (Shutty et al., 2026), which ensembles multiple perturbed matching decoders to approach near-optimal decoding accuracy.

Neural decoders learn decoding rules directly from data. Prior work has explored convolutional architectures (Meinerz et al., 2022), recurrent models (Varbanov et al., 2025), reinforcement-learning approaches (Sweke et al., 2021), graph neural networks (Lange et al., 2025), and Transformer-based models (Bausch et al., 2024). Hardware-oriented designs have further shown that compact neural models can reach sub-microsecond latency on FPGAs and other specialized accelerators (Overwater et al., 2022; Zhang et al., 2023; Senior et al., 2025).

Adaptive decoding addresses the mismatch that arises when hardware noise changes. Spitz et al. (2018) estimates matching weights from detector statistics in a sliding window. DGR (Wang et al., 2023) adapts surface-code decoding based on previous MWPM matchings. Bhardwaj et al. (2025) analytically estimate time-varying error rates from sliding windows of detector events. Sivak et al. (2024) use an RL-inspired optimization procedure to calibrate decoder-prior probabilities from experimental QEC data during calibration, rather than at runtime. Also during calibration, (Stein et al., 2026) adapts a lightweight neural model using externally provided hardware calibration features.

6 Conclusion

We presented SF-QEC, a two-timescale decoding framework that decouples runtime adaptivity from the latency-critical decoding path. SF-QEC adopts aligned slow and fast paths, combining a recurrent-transformer decoder on the fast path with a history-conditioned model on the slow path. Through hierarchical history embedding and recurrent-state fusion, SF-QEC improves the fast-path decoder in most settings across four IBM Phoenix patches and three detection-round configurations. Our results further show that the slow path can perform realistic history analysis and adaptation on commodity GPUs within the available latency budget and without requiring runtime labels. These findings suggest that history-conditioned two-timescale decoding is a promising direction for real-time adaptive QEC.

AI USE STATEMENT

Generative AI tools, including ChatGPT and Codex, assisted with research discussion, software development and debugging, experiment orchestration, result inspection, and drafting text and figures. Numerical results are derived from recorded experiments and saved prediction files, rather than generated measurements. The authors are responsible for verifying the AI-assisted code, citations, analyses, and final manuscript.

ETHICS STATEMENT

This study concerns quantum-memory measurements and synthetic detector data. It does not involve human subjects or personal data. We distinguish offline decoding from hardware error correction and report the limitations.

REPRODUCIBILITY STATEMENT

We provide the dataset, source code, model configurations, and analysis scripts used in this work. Section 4.1 and Appendices B-C specify the data acquisition, preprocessing, train/validation/test splits, model architectures, training settings, baseline implementations, and latency measurements.

REFERENCES

Johannes Bausch, Andrew W. Senior, et al. Learning high-accuracy error decoding for quantum processors. *Nature*, 2024. doi: 10.1038/s41586-024-08148-8.

Devansh Bhardwaj, Evangelia Takou, Yingjia Lin, and Kenneth R. Brown. Adaptive estimation of drifting noise in quantum error correction, 2025.

Sergey Bravyi, Martin Suchara, and Alexander Vargo. Efficient algorithms for maximum likelihood decoding in the surface code. *Physical Review A*, 90(3):032326, 2014. doi: 10.1103/PhysRevA.90.032326.

Sergey B Bravyi and A Yu Kitaev. Quantum codes on a lattice with boundary. *arXiv preprint quant-ph/9811052*, 1998.

Shane A. Caldwell, Moein Khazraee, et al. Platform architecture for tight coupling of high-performance computing with quantum processors. *arXiv preprint arXiv:2510.25213*, 2025.

Nicolas Delfosse and Naomi H. Nickerson. Almost-linear time decoding algorithm for topological codes. *Quantum*, 5:595, 2021. doi: 10.22331/q-2021-12-02-595.

Eric Dennis, Alexei Kitaev, Andrew Landahl, and John Preskill. Topological quantum memory. *arXiv preprint quant-ph/0110143*, 2001.

Austin G. Fowler. Optimal complexity correction of correlated errors in the surface code, 2013.

Craig Gidney. Stim: a fast stabilizer circuit simulator. *Quantum*, 5:497, 2021.

Oscar Higgott and Craig Gidney. Sparse blossom: Correcting a million errors per core second with minimum-weight matching. *Quantum*, 9:1600, 2025. doi: 10.22331/q-2025-01-20-1600.

Oscar Higgott, Thomas C. Bohdanowicz, Aleksander Kubica, Steven T. Flammia, and Earl T. Campbell. Improved decoding of circuit noise and fragile boundaries of tailored surface codes. *Physical Review X*, 13(3):031007, 2023. doi: 10.1103/PhysRevX.13.031007.

Timo Hillmann, Lucas Berent, Armanda O. Quintavalle, Jens Eisert, Robert Wille, and Joschka Roffe. Localized statistics decoding for quantum low-density parity-check codes. *Nature Communications*, 16:8214, 2025a. doi: 10.1038/s41467-025-63214-7.

Timo Hillmann, Lucas Berent, Armanda O. Quintavalle, Jens Eisert, Robert Wille, and Joschka Roffe. Localized statistics decoding for quantum low-density parity-check codes, 2025b. URL https://arxiv.org/abs/2406.18655.

IBM Quantum. Ibm quantum nighthawk r2. IBM Quantum Documentation, 2026. Accessed September 2026.

Sebastian Krinner, Nathan Lacroix, Ants Remm, Agustin Di Paolo, Elie Genois, Catherine Leroux, Christoph Hellings, Stefania Lazar, Francois Swiadek, Johannes Herrmann, et al. Realizing repeated quantum error correction in a distance-three surface code. *arXiv preprint arXiv:2112.03708*, 2021.

Moritz Lange, Pontus Havström, Basudha Srivastava, Isak Bengtsson, Valdemar Bergentall, Karl Hammar, Olivia Heuts, Evert van Nieuwenburg, and Mats Granath. Data-driven decoding of quantum error correcting codes using graph neural networks. *Physical Review Research*, 7(2): 023181, 2025. doi: 10.1103/PhysRevResearch.7.023181.

Kai Meinerz, Chae-Yeun Park, and Simon Trebst. Scalable neural decoder for topological surface codes. *Physical Review Letters*, 128(8):080505, 2022. doi: 10.1103/PhysRevLett.128.080505.

Ramon WJ Overwater, Masoud Babaie, and Fabio Sebastiano. Neural-network decoders for quantum error correction using surface codes: A space exploration of the hardware cost-performance tradeoffs. *IEEE Transactions on Quantum Engineering*, 3:1–19, 2022.

Alexandru Paler and Austin G. Fowler. Pipelined correlated minimum weight perfect matching of the surface code. *Quantum*, 7:1205, 2023. doi: 10.22331/q-2023-12-12-1205.

Joschka Roffe, David R. White, Simon Burton, and Earl Campbell. Decoding across the quantum low-density parity-check code landscape. *Physical Review Research*, 2(4):043423, 2020a. doi: 10.1103/PhysRevResearch.2.043423.

Joschka Roffe, David R. White, Simon Burton, and Earl T. Campbell. Decoding across the quantum LDPC code landscape, 2020b. URL `https://arxiv.org/abs/2005.07016`.

Andrew W. Senior, Thomas Edlich, Francisco J. H. Heras, Lei M. Zhang, Oscar Higgott, James S. Spencer, Taylor Applebaum, Sam Blackwell, Justin Ledford, Akvilė Žemgulytė, Augustin Žídek, Noah Shutty, Andrew Cowie, Yin Li, George Holland, Peter Brooks, Charlie Beattie, Michael Newman, Alex Davies, Cody Jones, Sergio Boixo, Hartmut Neven, Pushmeet Kohli, and Johannes Bausch. A scalable and real-time neural decoder for topological quantum codes, 2025.

Noah Shutty, Michael Newman, and Benjamin Villalonga. Efficient near-optimal decoding of the surface code through ensembling. *Physical Review Letters*, 136(7):070603, 2026. doi: 10.1103/77j6-32xx.

Volodymyr Sivak, Michael Newman, and Paul Klimov. Optimization of decoder priors for accurate quantum error correction. *Physical Review Letters*, 133(15):150603, 2024. doi: 10.1103/PhysRevLett.133.150603.

Stephen T. Spitz, Brian Tarasinski, Carlo W. J. Beenakker, and Thomas E. O'Brien. Adaptive weight estimator for quantum error correction in a time-dependent environment. *Advanced Quantum Technologies*, 1(1):1800012, 2018. doi: 10.1002/qute.201800012.

Samuel Stein, Shuwen Kan, Chenxu Liu, Adrian Harkness, Sean Garner, Zefan Du, Yufei Ding, Ying Mao, and Ang Li. Calibration-conditioned FiLM decoders for low-latency decoding of quantum error correction evaluated on IBM repetition-code experiments, 2026.

Ryan Sweke, Markus S. Kesselring, Evert P. L. van Nieuwenburg, and Jens Eisert. Reinforcement learning decoders for fault-tolerant quantum computation. *Machine Learning: Science and Technology*, 2(2):025005, 2021. doi: 10.1088/2632-2153/abc609.

Barbara M Terhal. Quantum error correction for quantum memories. *Reviews of Modern Physics*, 87(2):307–346, 2015.

Boris M. Varbanov, Marc Serra-Peralta, David Byfield, and Barbara M. Terhal. Neural network decoder for near-term surface-code experiments. *Physical Review Research*, 7(1):013029, 2025. doi: 10.1103/PhysRevResearch.7.013029.

Hanrui Wang, Pengyu Liu, Yilian Liu, Jiaqi Gu, Jonathan M. Baker, Frederic T. Chong, and Song Han. DGR: Tackling drifted and correlated noise in quantum error correction via decoding graph re-weighting, 2023.

Mengyu Zhang, Xiangyu Ren, Guanglei Xi, Zhenxing Zhang, Qiaonian Yu, Fuming Liu, Hualiang Zhang, Shengyu Zhang, and Yi-Cong Zheng. A scalable, fast and programmable neural decoder for fault-tolerant quantum computation using surface codes. *arXiv preprint arXiv:2305.15767*, 2023.

A ACQUISITION OF FIGURE 1

Figure 1 uses surface-code memory experiments performed on IBM Phoenix superconducting quantum computer. We analyze the top-left distance-3 surface-code patch comprising nine data qubits and eight syndrome-measurement ancillas in Figure A.1. Each shot implements a logical-Z memory experiment initialized in $|0_L\rangle$, with 3 rounds of syndrome extraction followed by measurement of the data qubits in the Z basis. The circuit uses a reset-free protocol: ancillas are initialized at the beginning of each shot but are not reset between syndrome rounds. The figure uses three successive acquisition batches, numbered 13-15, containing 10,000 shots each for this circuit condition. The same compiled circuit and physical-qubit placement are reused across batches. All 30,000 shots are included.

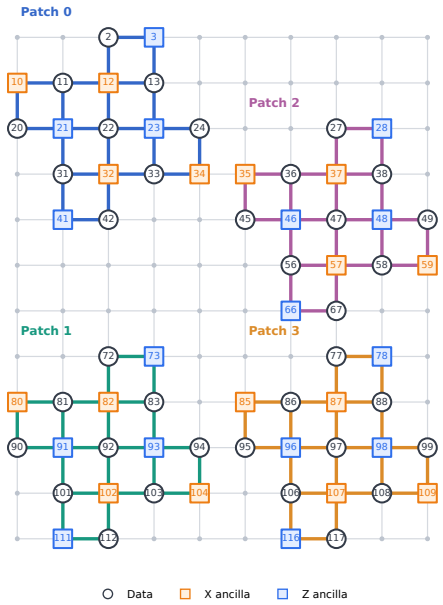


Figure A.1: Patches used on IBM Phoenix.

Decoding is performed offline with a two-MWPM-pass correlated matching approach described in Shutty et al. (2026). All three batches are decoded using the original calibration-derived DEM associated with batch-11. The DEM contains 219 error mechanisms, whose correlations are retained when constructing the matching representation. During decoding, the first pass matches using edge-like mechanisms with weights $-\ln p$. Selected edges determine conditional probability updates for complementary components of correlated error mechanisms. A second matching pass produces the logical-flip prediction. The implementation uses a modified PyMatching backend to reproduce this procedure.

Within each batch, we compute each LER data point with the following:

$$\widehat{p}_{b,k} = \frac{1}{2000} \sum_{i=500k}^{500k+1999} e_{b,i}, \qquad k = 0, \ldots, 16. \tag{A.1}$$

where $e_{b,i} \in \{0, 1\}$ is the error of the i-th shot in batch b. This gives 17 windows per batch and 51 estimates overall, effectively, 2000-shot rolling window with 500-step stride. Windows do not cross batch boundaries. Shading represents pointwise 95% Wilson binomial confidence intervals.

B ADDITIONAL EXPERIMENTAL DETAILS

B.1 ACQUISITION, PATCHES, AND DETECTOR REPRESENTATION

All patches have the same canonical distance-3 geometry, logical-Z observable, and detector convention, but different physical placements and calibration-derived DEM probabilities. The same compiled circuit and patch placement are reused across batches. Physical IDs are used to bind calibration values to circuit locations; canonical stabilizer coordinates supply the neural spatial features. Thus a change of patch does not require a different lattice architecture, but does require its own data and noise prior. The memory experiment starts from the logical-zero preparation, or $|0_L\rangle$ as mentioned in Appendix A and ends with destructive data-qubit readout. Decoded corrections are not applied to the device during these acquisitions. The exported detector vector contains $8r$ bits: four initial Z checks, $8(r-1)$ intermediate checks, and four final-boundary checks; initial X checks are omitted because they are random under logical-Z preparation. For the reset-free circuit, the final boundary is corrected against the last stabilizer eigenvalue, and the same detector-coordinate transformation is applied consistently to the experimental data and DEM supports. AQ2-RT inputs use the corresponding gauge-fixed signed encoding $1 - 2b$, with unavailable initial/final X entries masked. Batch order follows acquisition order. Within each batch, the dataset preserves the returned record order. History windows are causal with respect to the stored record sequence.

Table B.1: Acquisition timing. Start and end are IBM Runtime job timestamps. For batch i, start-to-start is measured from the start of batch $i-1$, and end-to-start from its end. All values are rounded to the nearest second; intervals are calculated before rounding. †: batch 16 is retained here for the timeline but excluded from model training and evaluation.

Batch	Start (hh:mm:ss)	End (hh:mm:ss)	Start-to-start (mm:ss)	End-to-start (mm:ss)
1	11:46:07	11:46:27	–	–
2	12:20:33	12:20:55	34:26	34:06
3	12:55:44	12:55:55	35:12	34:50
4	13:45:16	13:46:25	49:31	49:20
5	14:21:25	14:41:19	36:09	35:00
6	15:11:56	15:12:20	50:31	30:37
7	15:47:09	15:47:21	35:13	34:49
8	16:22:21	16:22:35	35:12	35:00
9	16:57:33	16:57:46	35:12	34:58
10	17:32:44	17:33:02	35:11	34:58
11	18:07:57	18:08:12	35:13	34:55
12	18:43:10	18:43:21	35:13	34:58
13	19:18:23	19:18:34	35:13	35:02
14	19:59:26	19:59:41	41:04	40:53
15	20:42:15	20:43:27	42:49	42:34
16†	21:16:46	21:17:07	34:31	33:19
17	21:51:22	21:52:50	34:36	34:15
18	22:31:57	22:33:12	40:34	39:07
19	23:13:25	23:13:40	41:29	40:14
20	23:53:00	23:53:33	39:34	39:19

Table B.1 lists the timing of all 20 batches. The acquisition schedule specified a 30-minute cooldown. The recorded start-to-start intervals have a median of 35 min 13 s and range from 34 min 26 s to 50 min 31 s. The first and last batch starts are approximately 12.1 hours apart. The eight test batches (12–15 and 17–20) span approximately 5.2 hours from the start of batch 12 to the start of batch 20. Excluding batch 16 leaves a 69 min 7 s start-to-start interval between batches 15 and 17.

B.2 Fixed calibration prior and synthetic sampling

For each patch and round count, we use the original, unoptimized batch-11 DEM as the fixed anchor prior. Its calibration snapshot was retrieved immediately after the corresponding job and is the closest retained calibration record, rather than an exact reconstruction of the noise during execution. The DEM is constructed by mapping each reported CZ-gate error to a two-qubit depolarizing channel and each reported readout error to a premeasurement bit-flip channel; idle and reset errors are not included.

Let A and $\boldsymbol{l}$ denote the detector and logical incidence of the DEM mechanisms, with mechanism probabilities p_j. Synthetic samples are generated by drawing $z_j \sim \mathrm{Bernoulli}(p_j)$ and computing

$$\boldsymbol{v} = A\boldsymbol{z} \pmod 2, \qquad \ell = \boldsymbol{l}^{\mathsf{T}}\boldsymbol{z} \pmod 2. \tag{B.1}$$

v is the vector of detection event bits for one shot. Complete correlated detector and logical supports of each DEM mechanism are retained. No synthetic drift is introduced: training draws fresh samples from this fixed prior each epoch, while validation and test sets use separate fixed samples.

B.3 Data splits and training

Table B.2 gives the exact target ranges; all indices are zero-based and half-open. Training and validation are split within batches 1–11, whereas test batches occur later in the acquisition sequence. For each of batches 1–11, we use 8,000 shots to update the model and the remaining 2,000 shots to select checkpoints and determine early stopping.

For a target block of shots $[s, s+20)$, the slow encoder receives only detector records in $[s-500, s)$. Its two output memories are reused for all 20 target shots on the fast path. Windows never cross batch

Table B.2: Experimental target splits, separately for each patch/condition.

Use	Batches	Target rows per batch	Total
Fast / baseline training	1–11	$[0, 8000)$	88,000
Joint SF-QEC training	1–11	$[500, 8000)$	82,500
Validation	1–11	$[8000, 10000)$	22,000
Earlier test subset	12–15	$[2000, 10000)$	32,000
Later test subset	17–20	$[2000, 10000)$	32,000

boundaries and have no padding or future inputs. The first 500 training records provide history and are therefore not joint targets; their labels have already been used by the stage-2 fast model. At each training epoch, we shuffle the order of the training blocks, but keep the shots within each 500-shot history in their original order. During validation and testing, the slow model reads detector records from preceding shots in the same batch. It does not receive the logical labels of those history shots. All test methods score the same last 8,000 records, even though SF-QEC requires only a 500-shot history. Each joint epoch has 4,125 history/target blocks.

The three SF-QEC stages are specified in Table B.3. Each selects the checkpoint with minimum validation BCE, breaking an exact BCE tie by validation error count. Epoch zero is eligible; a harmful fine-tune can retain its initialization. Early stopping uses five consecutive nonimproving epochs, with no minimum epoch requirement. The epoch cap applies separately to each stage. For each combination of fast- and slow-model sizes, we start with a new slow model and new layers that combine information from the two models. Combinations with the same fast-model size start from the same fast decoder trained in the first two stages. We then train each combination separately, without carrying over changes learned by another combination.

Table B.3: SF-QEC training hyperparameters for 3 training stages. Stage 3 updates both networks and the fusion layers from its first optimizer step.

Setting	Synthetic fast	Experimental fast	Joint fast + slow
Initialization	Random fast	Selected synthetic	Selected fast and new slow
Examples / epoch	2,097,152	88,000	82,500 targets
Optimizer	Lion	AdamW	AdamW
Learning rate	2×10^{-5}	10^{-5}	10^{-5} fast and 10^{-4} slow
Minibatch	1,024 shots	64 shots	8 histories $\times$ 20 targets
Updates / epoch	2,048	1,375	516
Validation	100,000 synthetic	22,000 experimental	Same 22,000 experimental
Epoch cap / patience	50 / 5	50 / 5	50 / 5

All three stages use FP32, weight decay 10^{-4}, gradient-norm clipping at 5, and ReduceLROnPlateau on validation BCE (factor 0.5, patience 3, minimum learning rate 10^{-7}). Lion uses $(\beta_1, \beta_2) = (0.9, 0.99)$; synthetic training warms up linearly over the first 1,000,000 presented examples. The fast-input masking probabilities are 0.8 per example and 0.5 per stabilizer, with masking disabled at evaluation; the slow history is not masked. The maximum synthetic budget per fast model is 104,857,600 shots, reduced when early stopping occurs. Experimental epochs revisit the same examples in shuffled order. We use one seed, 835468629, deterministic PyTorch algorithms, and disabled TF32. Checkpoints retain optimizer, scheduler, random number generator, and early-stopping state for interruption recovery.

We exclude batch 16 from all comparisons. This decision was made after reviewing earlier experimental results; it was not based on a hardware-failure rule defined before the experiments. We had also examined the other test batches in earlier experiments. In the current runs, we choose checkpoints using validation results only and save predictions before comparing them with the test labels. However, these steps do not make the test data new to the study. Excluding batch 16 and using only one training seed limit what we can conclude about performance on other data.

We also examined the detector records without using logical labels and found unusual detector-event rates in batch 8 across all 12 combinations of patch and round count. We kept batch 8 in training

and validation. For nine rounds, after correcting the detector representation, the percentages of detector bits equal to 1 in batch 16 are 39.02%, 31.16%, 34.04%, and 39.91% for patches 0–3, respectively. The corresponding median percentages across the other batches, excluding batches 8 and 16, are 12.77%, 9.05%, 11.50%, and 15.09%. These differences show that the detector statistics substantially changed.

B.4 Model configurations and comparison controls

The fast decoder uses eight stabilizer tokens and two gated recurrent blocks, with two spatial transformer layers between them and two more before readout. Spatial attention uses four heads and two-dimensional rotary position encoding based on canonical stabilizer coordinates. The recurrent gates use four blocks. Logical readout uses two cross-attention blocks, each with eight heads of dimension 32, followed by two residual gated dense blocks. Dense blocks use an internal width four times the embedding width.

The fast decoder groups syndrome frames in sets of six and processes the final boundary separately. This gives $\lceil r/6 \rceil + 1$ recurrent steps per shot: two for $r = 3, 6$ and three for $r = 9$. Its memories reset at each shot. The slow encoder projects all $r + 1$ frames within each shot, then groups every 20 shots into one input block. It processes the resulting 25 history blocks using two recurrent blocks separated by two spatial transformer layers. Its final block outputs only its memory, without a feature-output projection.

The slow encoder supplies the final memories from both recurrent blocks. Each is concatenated with the corresponding updated fast memory and passed through a linear layer that maps $w_f + w_s$ features to w_f, where w_f and w_s are the fast and slow embedding widths. These two layers are shared across stabilizers and recurrent steps. They initially copy the fast memory and give zero weight to the slow memory, with zero bias. We check that this reproduces the trained fast decoder before joint training. During evaluation, weights stay fixed and only the cached slow memories change. The cache contains $2 \times 8 \times w_s$ values: 4 or 8 KiB in FP32 for slow widths 64 or 128.

Table B.4: Parameter counts. Slow-model size depends on r; fast-model size does not. Fusion parameters are listed separately.

Component	$r = 3$	$r = 6$	$r = 9$
Fast, width 32	199,329	199,329	199,329
Fast, width 64	653,121	653,121	653,121
Fast, width 128	2,328,705	2,328,705	2,328,705
Slow, width 64	253,312	265,600	277,888
Slow, width 128	1,006,336	1,055,488	1,104,640
Fusion, $(w_f, w_s) = (32, 64)$		6,208	
Fusion, $(32, 128)$		10,304	
Fusion, $(64, 64)$		16,512	
Fusion, $(64, 128)$		24,704	
Fusion, $(128, 128)$		65,792	

At each round count, patch 0 covers all five width pairings in Table B.4; patches 1–3 cover only $(64, 64)$ and $(128, 128)$. Each patch uses its own trained fast decoder.

Comparing SF-QEC with its stage-2 fast decoder includes the effect of the additional joint training. To examine how the trained model uses history, we also keep its weights fixed and replace the slow context with zeros, the mean context from training, the first context in each test batch, or context computed after shuffling the shots in each history. These tests change the inputs to an existing model; they are not separately trained baselines. The replacements can also differ from the contexts seen during training.

B.5 Classical baseline implementations

MWPM uses PyMatching (Higgott & Gidney, 2025). Union-find uses unweighted growth and peeling (Delfosse & Nickerson, 2021). CM uses our implementation of the two-pass correlated matching

procedure in Shutty et al. (2026), with the original DEM and no ensemble. It does not use PyMatching's built-in correlation option. Graph-based comparisons apply the reset-free boundary correction described above. Earlier graph-projection results with conflicting logical edge assignments are excluded from decoder rankings.

BP–OSD and BP–LSD use the detector–mechanism incidence matrix (Roffe et al., 2020b; Hillmann et al., 2025b). Both use minimum-sum belief propagation with scaling 0.625 and a serial update schedule. Validation selects from 10, 30, or 100 BP iterations and postprocessing orders 0 or 2. DGR uses past matching results to update edge probabilities and applies the heuristic correlation rule of Wang et al. (2023). We also report an alignment-only version that updates edge probabilities without this extra rule. DGR and the syndrome-moment sliding-window decoder use 2,000 preceding shots, update every 1,000 shots, and reset at each batch.

B.6 Evaluation, latency, and reproducibility

AQ2-RT and SF-QEC predictions use a probability threshold of 0.5. We count each of the 64,000 test targets once when computing the overall LER. Wilson intervals describe uncertainty across shots; they do not account for correlations between shots or variation across training seeds.

Training uses A100 GPUs with 40 or 80 GB memory and H100 GPUs with 80 GB memory, with one GPU per model. The patch-0 experiments use A100-SXM4-80GB GPUs. The training environment uses Python 3.11.13, PyTorch 2.5.1, CUDA 12.4, Stim 1.15.0, and NumPy 2.0.1.

Latency is measured on A100-SXM4-80GB and GH200 using the same six patch-0 checkpoints: $r = 3, 6, 9$ at widths 64/64 and 128/128. Each run uses one otherwise idle GPU, four CPU threads, deterministic execution, and disabled TF32. The A100 uses the training software environment. GH200 uses an AArch64 host, Python 3.11.13, PyTorch 2.5.1.post206, CUDA 12.6, NumPy 2.0.1, Triton 3.1.0, and driver 570.211.01. The GPU comparison therefore also includes software differences. FP16 uses CUDA autocast with the same FP32-trained weights.

All inputs and cached context are already on the GPU. Fast latency covers one complete shot, including detector conversion, the fast network, both fusion layers, and logical readout. Slow latency covers both context memories computed from one 500-shot history. We compile fixed input shapes with Inductor and constant-weight freezing, then use CUDA graph replay. We retain ordinary graph replay if compilation fails numerical checks. The $r = 9$, width-64 fast FP16 case uses `emulate_precision_casts=True` on both GPUs to preserve intermediate rounding.

Ten warm-up calls precede graph capture. We check 32 input examples against uncompiled execution with absolute and relative tolerances of 3×10^{-3}; all fast-path binary predictions must agree. On GH200, the 32 FP32 predictions also match the saved reference logits within 2×10^{-5}, with no changed decisions. After 3,000 untimed replays, we run seven timing trials, each containing 200 fast calls or 50 slow calls. We report the median of the seven mean per-call times measured with CUDA events. Appendix tables C.6 and C.7 identify the execution method used for each result.

These are steady-state GPU timings. They exclude compilation, graph capture, acquisition, data transfers, scheduling, and communication between paths. Fast timing also excludes slow computation. A slow refresh every 20 shots costs $T_{\text{slow}}/20$ per shot on average, but we do not demonstrate an overlapping execution schedule or a hardware deadline. Accuracy evaluation assumes context is ready at each 20-shot block boundary and does not simulate delivery delays or queues.

C Additional results

All experimental results use 64,000 test shots per setting unless stated otherwise. Each model is trained with one random seed. In the tables, a single width denotes AQ2-RT after fine-tuning on experimental data; a fast/slow width pair denotes SF-QEC.

C.1 Model widths and training epochs

Table C.1 reports LER for batches 12–15 and 17–20, each with 32,000 test shots, and for both groups combined. Epoch counts refer to the final training stage. The selected epoch gives the saved model;

completed epochs include training after that epoch until stopping. Epoch 0 means the model before this stage was selected.

At $r = 9$, patch 3, the 128/128 SF-QEC model makes 13,023 errors compared with 12,953 for AQ2-RT, an increase of 70. Although epoch 0 was eligible, epoch 8 was selected using validation loss on batches 1–11.

Table C.1: LER (%) for all model widths. Epochs: selected / completed, for AQ2-RT fine-tuning or SF-QEC joint training.

r	Patch	Fast / slow	Epochs	B12–15	B17–20	All
3	0	32	41 / 46	11.434	10.681	11.058
3	0	32 / 128	5 / 10	11.256	10.694	10.975
3	0	32 / 64	15 / 20	11.269	10.769	11.019
3	0	64	22 / 27	11.459	10.881	11.170
3	0	64 / 128	13 / 18	11.266	10.688	10.977
3	0	64 / 64	13 / 18	11.088	10.666	10.877
3	0	128	29 / 34	11.406	10.441	10.923
3	0	128 / 128	8 / 13	11.012	10.441	10.727
3	1	64	23 / 28	3.731	3.612	3.672
3	1	64 / 64	7 / 12	3.434	3.428	3.431
3	1	128	13 / 18	3.797	3.716	3.756
3	1	128 / 128	5 / 10	3.356	3.403	3.380
3	2	64	31 / 36	6.922	6.425	6.673
3	2	64 / 64	12 / 17	6.628	6.272	6.450
3	2	128	21 / 26	6.944	6.359	6.652
3	2	128 / 128	7 / 12	6.628	6.353	6.491
3	3	64	28 / 33	8.034	7.938	7.986
3	3	64 / 64	17 / 22	7.622	7.631	7.627
3	3	128	26 / 31	8.097	7.909	8.003
3	3	128 / 128	16 / 21	7.728	7.594	7.661
6	0	32	45 / 50	21.106	20.594	20.850
6	0	32 / 128	9 / 14	20.988	20.466	20.727
6	0	32 / 64	8 / 13	21.044	20.391	20.717
6	0	64	41 / 46	20.884	20.125	20.505
6	0	64 / 128	8 / 13	20.600	19.900	20.250
6	0	64 / 64	8 / 13	20.525	19.863	20.194
6	0	128	27 / 32	20.488	19.884	20.186
6	0	128 / 128	3 / 8	20.275	19.766	20.020
6	1	64	28 / 33	8.128	7.912	8.020
6	1	64 / 64	9 / 14	7.794	7.737	7.766
6	1	128	32 / 37	7.944	7.753	7.848
6	1	128 / 128	8 / 13	7.731	7.641	7.686
6	2	64	41 / 46	15.522	14.078	14.800
6	2	64 / 64	5 / 10	15.250	14.034	14.642
6	2	128	26 / 31	14.866	13.984	14.425
6	2	128 / 128	8 / 13	14.616	13.884	14.250
6	3	64	34 / 39	15.066	14.231	14.648
6	3	64 / 64	7 / 12	14.672	13.966	14.319
6	3	128	29 / 34	14.847	14.113	14.480
6	3	128 / 128	10 / 15	14.647	13.900	14.273
9	0	32	26 / 31	29.094	29.653	29.373
9	0	32 / 128	12 / 17	28.825	29.422	29.123
9	0	32 / 64	12 / 17	28.878	29.488	29.183
9	0	64	14 / 19	28.375	29.078	28.727
9	0	64 / 128	4 / 9	28.337	28.663	28.500
9	0	64 / 64	4 / 9	28.312	28.597	28.455
9	0	128	14 / 19	27.694	28.537	28.116
9	0	128 / 128	10 / 15	27.475	28.100	27.787
9	1	64	37 / 42	12.319	12.378	12.348
9	1	64 / 64	8 / 13	12.172	12.081	12.127
9	1	128	19 / 24	12.166	12.291	12.228
9	1	128 / 128	9 / 14	11.953	11.916	11.934
9	2	64	29 / 34	22.209	20.997	21.603

r	Patch	Fast / slow	Epochs	B12–15	B17–20	All
9	2	64 / 64	0 / 5	22.209	20.997	21.603
9	2	128	15 / 20	21.834	20.688	21.261
9	2	128 / 128	5 / 10	21.397	20.384	20.891
9	3	64	23 / 28	21.384	20.231	20.808
9	3	64 / 64	15 / 20	21.044	20.238	20.641
9	3	128	21 / 26	20.663	19.816	20.239
9	3	128 / 128	8 / 13	20.806	19.891	20.348

C.2 Effect of changing the slow context

Table C.2 compares rolling context with the four replacements described in Appendix B.4: zero, mean training context, first context in each test batch, and shuffled history. All model weights remain fixed; these tests measure the effect of changing context without retraining.

Table C.2: LER (%) with different slow contexts for the same 33 trained SF-QEC models.

r	Patch	Fast / slow	Rolling	Zero	Mean	First	Shuffle
3	0	32 / 128	10.975	11.047	11.111	11.016	10.908
3	0	32 / 64	11.019	11.069	11.220	11.053	10.988
3	0	64 / 128	10.977	11.155	11.256	11.084	10.941
3	0	64 / 64	10.877	11.186	11.233	11.064	10.875
3	0	128 / 128	10.727	11.098	11.003	10.875	10.706
3	1	64 / 64	3.431	3.573	3.653	3.447	3.430
3	1	128 / 128	3.380	3.639	3.644	3.372	3.389
3	2	64 / 64	6.450	6.669	6.898	6.528	6.511
3	2	128 / 128	6.491	6.684	6.950	6.500	6.497
3	3	64 / 64	7.627	7.986	7.888	8.062	7.713
3	3	128 / 128	7.661	8.194	7.948	8.012	7.688
6	0	32 / 128	20.727	20.906	20.881	20.803	20.698
6	0	32 / 64	20.717	20.922	20.913	20.944	20.777
6	0	64 / 128	20.250	20.478	20.591	20.327	20.212
6	0	64 / 64	20.194	20.448	20.545	20.227	20.195
6	0	128 / 128	20.020	20.169	20.217	20.084	20.028
6	1	64 / 64	7.766	7.828	7.948	7.758	7.787
6	1	128 / 128	7.686	7.750	7.872	7.622	7.661
6	2	64 / 64	14.642	14.756	14.722	14.675	14.630
6	2	128 / 128	14.250	14.355	14.406	14.275	14.250
6	3	64 / 64	14.319	14.569	14.502	14.588	14.338
6	3	128 / 128	14.273	14.552	14.464	14.478	14.367
9	0	32 / 128	29.123	29.322	29.363	29.298	29.198
9	0	32 / 64	29.183	29.350	29.375	29.342	29.192
9	0	64 / 128	28.500	28.747	28.688	28.547	28.572
9	0	64 / 64	28.455	28.756	28.711	28.592	28.458
9	0	128 / 128	27.788	28.194	28.094	28.003	27.773
9	1	64 / 64	12.127	12.188	12.184	12.144	12.113
9	1	128 / 128	11.934	12.073	12.098	11.989	11.903
9	2	64 / 64	21.603	21.603	21.603	21.603	21.603
9	2	128 / 128	20.891	21.181	21.066	21.005	20.877
9	3	64 / 64	20.641	20.770	20.741	20.956	20.603
9	3	128 / 128	20.348	20.389	20.447	20.475	20.342

Averaged equally over the 12 patch/round settings, replacing rolling context with zero, mean training context, or first context increases LER by 0.184, 0.208, and 0.126 percentage points for 64/64, and by 0.236, 0.230, and 0.104 percentage points for 128/128. Shuffling changes the averages by only 0.010 and 0.003 percentage points, respectively. Thus, average LER is more sensitive to replacing recent context than to changing the order of shots within it.

C.3 ROLLING-WINDOW AND PER-BATCH RESULTS

Figures C.1–C.24 compare SF-QEC-64 and SF-QEC-128 with AQ2-RT across $r \in \{3, 6, 9\}$ and patches 0–3. Each figure shows batches 12–15 in the first row and 17–20 in the second. Orange dashed curves show AQ2-RT; blue solid curves show SF-QEC. Both use FP32 with fixed weights.

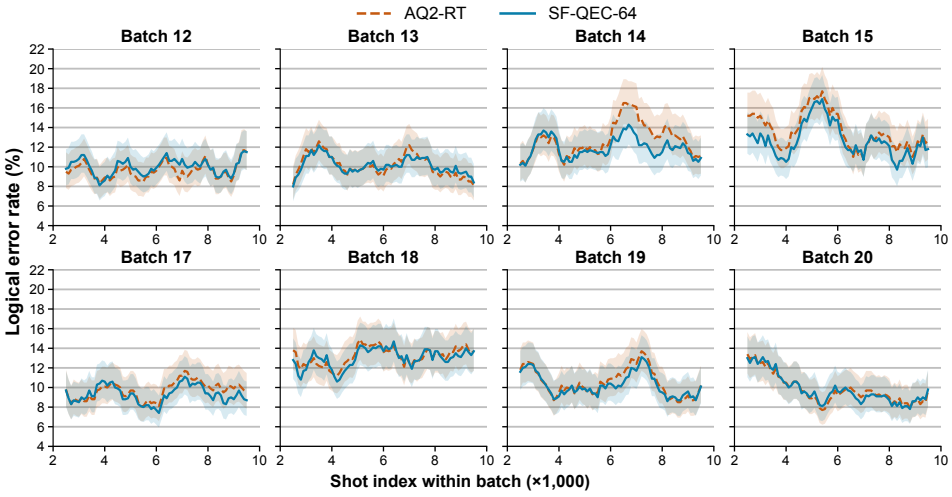


Figure C.1: Rolling-window LER at $r = 3$, patch 0, width 64.

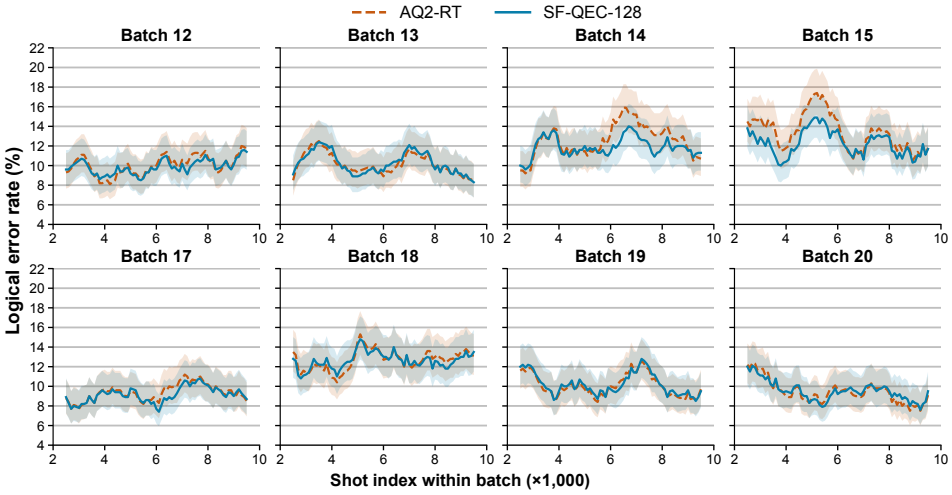


Figure C.2: Rolling-window LER at $r = 3$, patch 0, width 128.

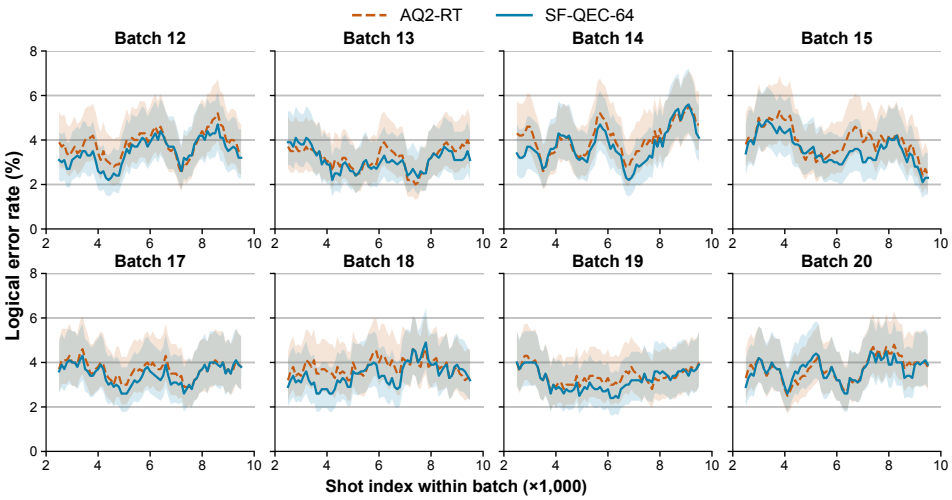


Figure C.3: Rolling-window LER at $r = 3$, patch 1, width 64.

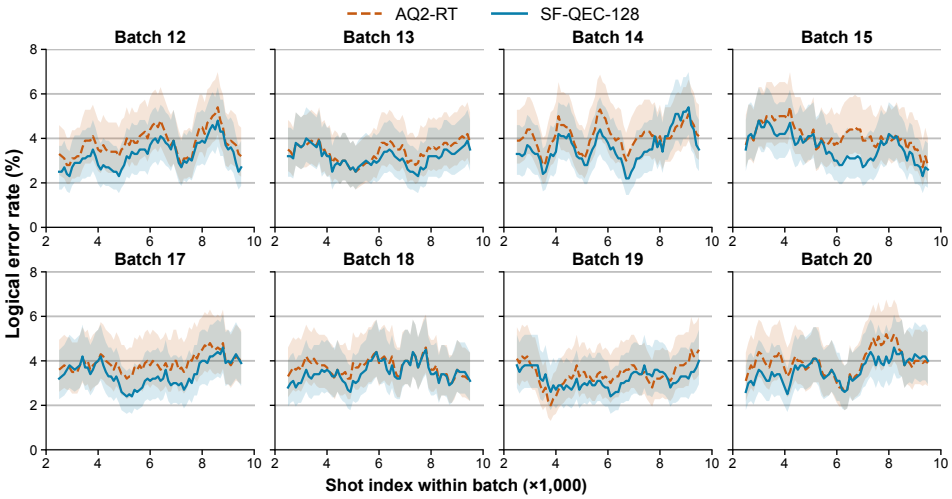


Figure C.4: Rolling-window LER at $r = 3$, patch 1, width 128.

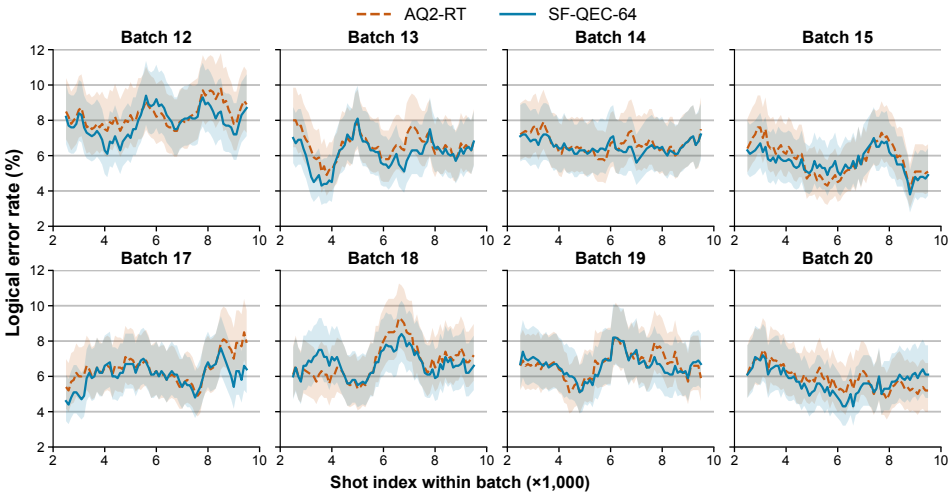


Figure C.5: Rolling-window LER at $r = 3$, patch 2, width 64.

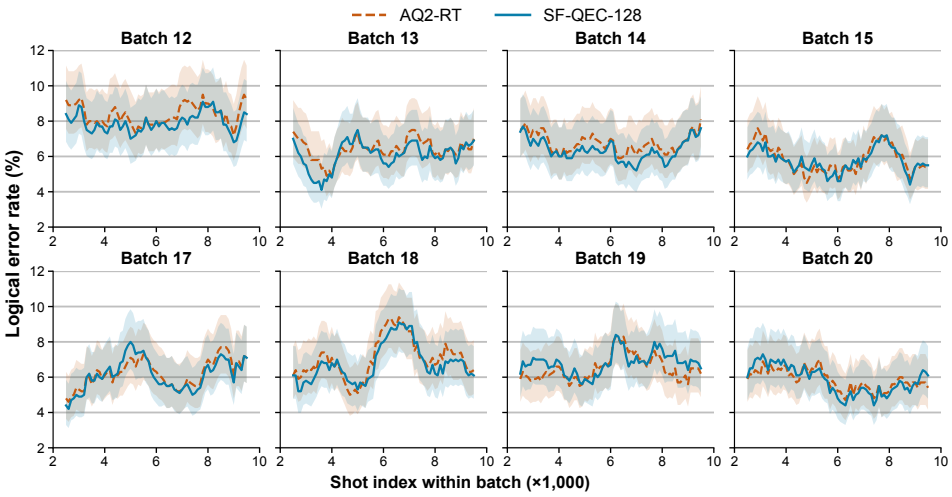


Figure C.6: Rolling-window LER at $r = 3$, patch 2, width 128.

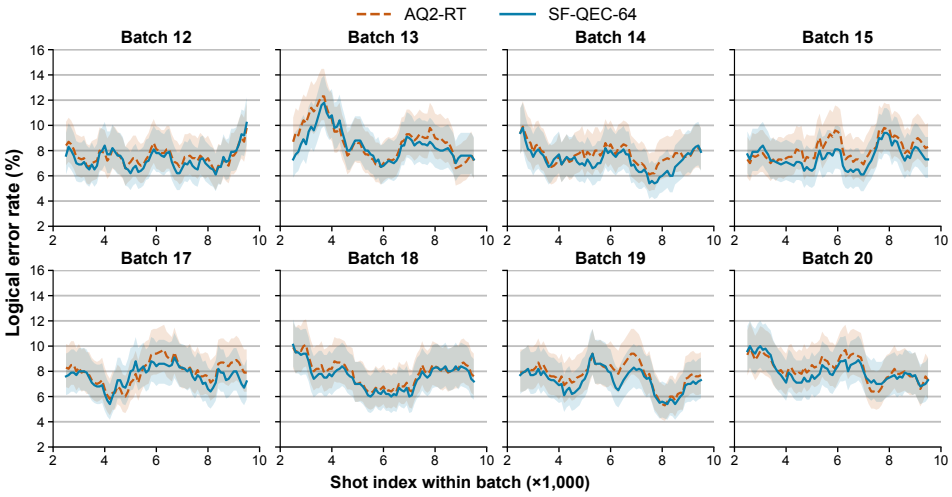


Figure C.7: Rolling-window LER at $r = 3$, patch 3, width 64.

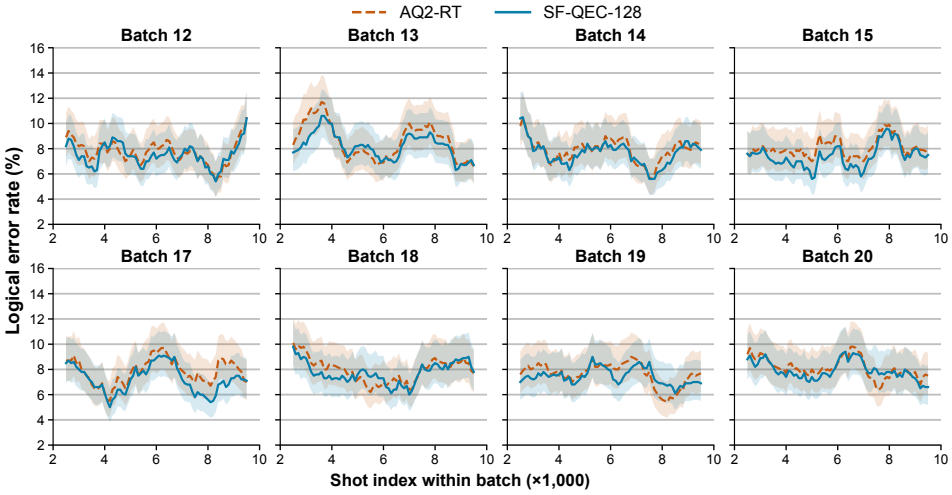


Figure C.8: Rolling-window LER at $r = 3$, patch 3, width 128.

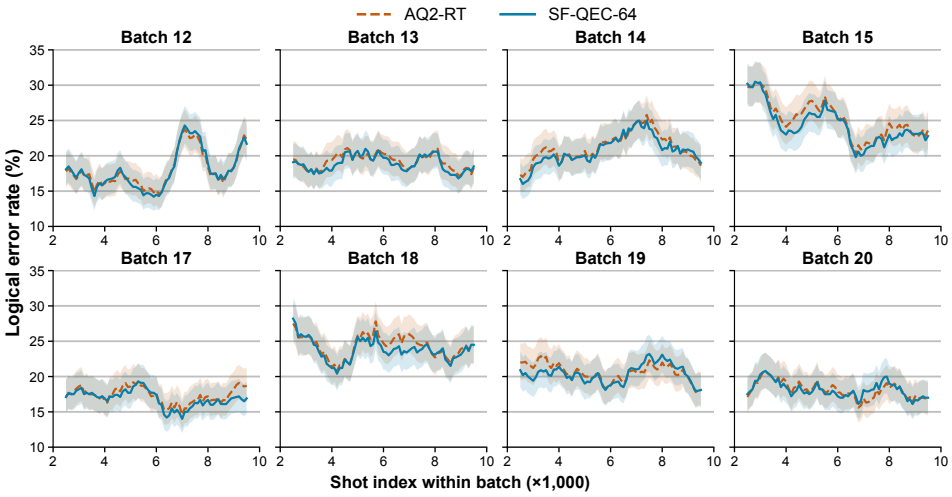


Figure C.9: Rolling-window LER at $r = 6$, patch 0, width 64.

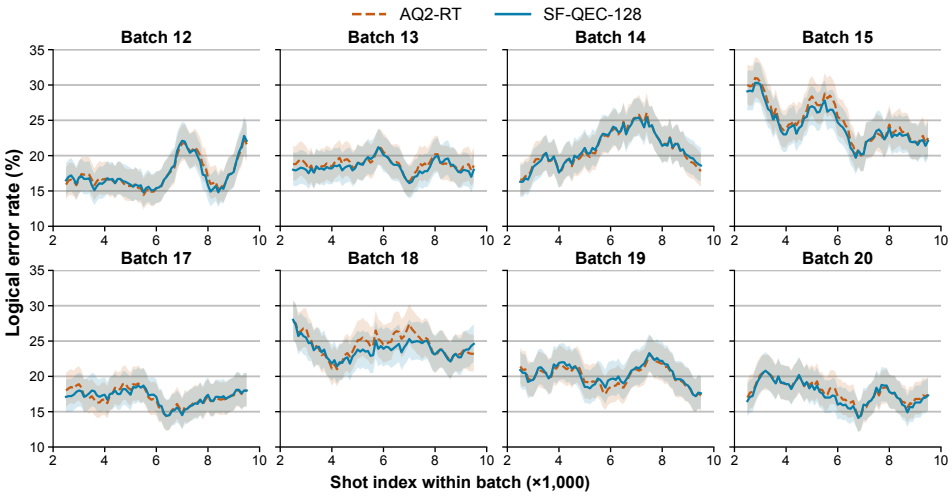


Figure C.10: Rolling-window LER at $r = 6$, patch 0, width 128.

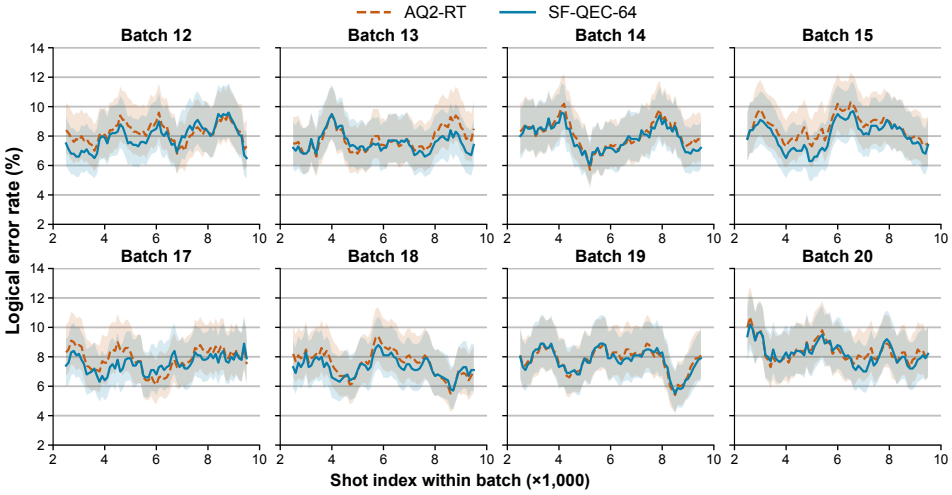


Figure C.11: Rolling-window LER at $r = 6$, patch 1, width 64.

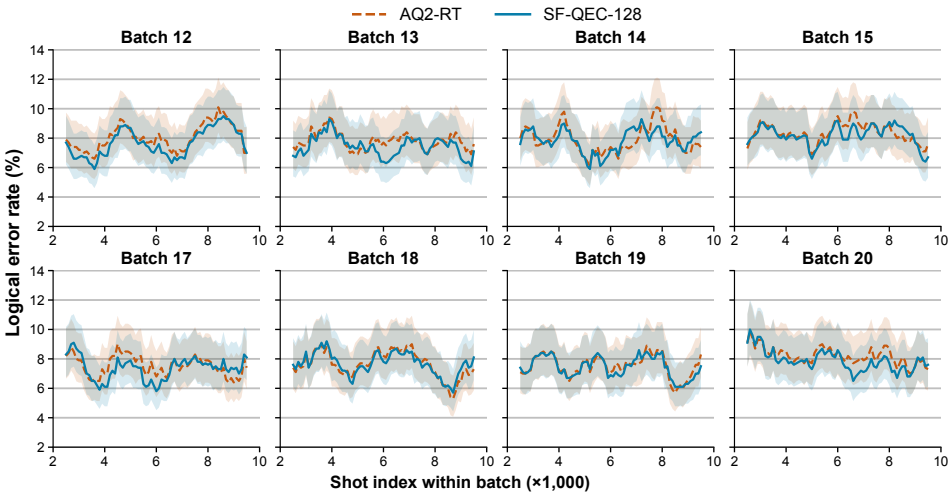


Figure C.12: Rolling-window LER at $r = 6$, patch 1, width 128.

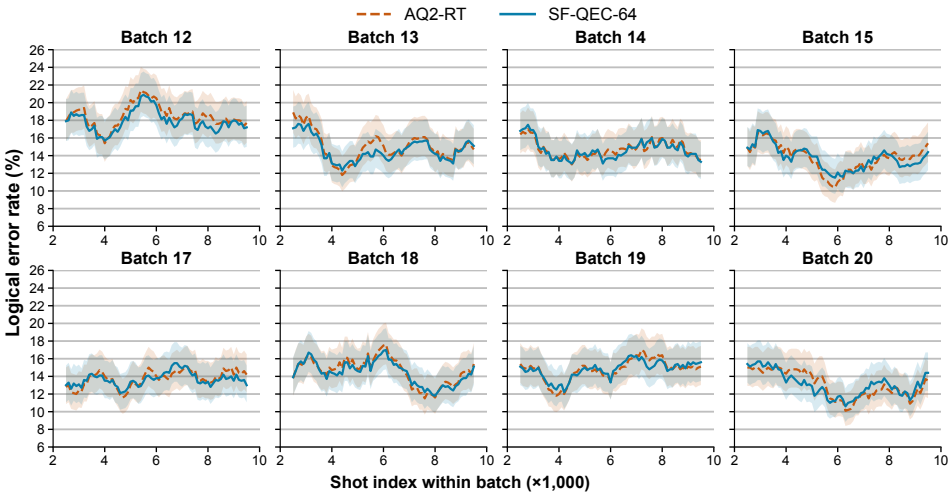


Figure C.13: Rolling-window LER at $r = 6$, patch 2, width 64.

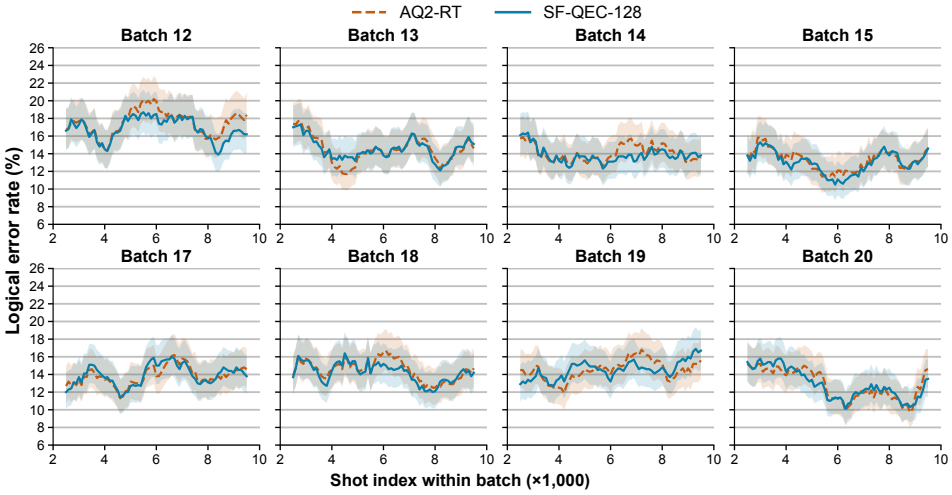


Figure C.14: Rolling-window LER at $r = 6$, patch 2, width 128.

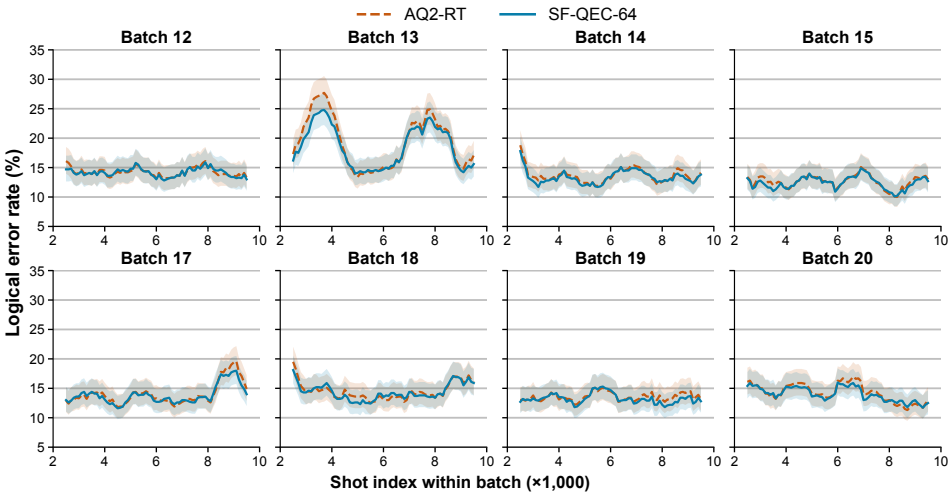


Figure C.15: Rolling-window LER at $r = 6$, patch 3, width 64.

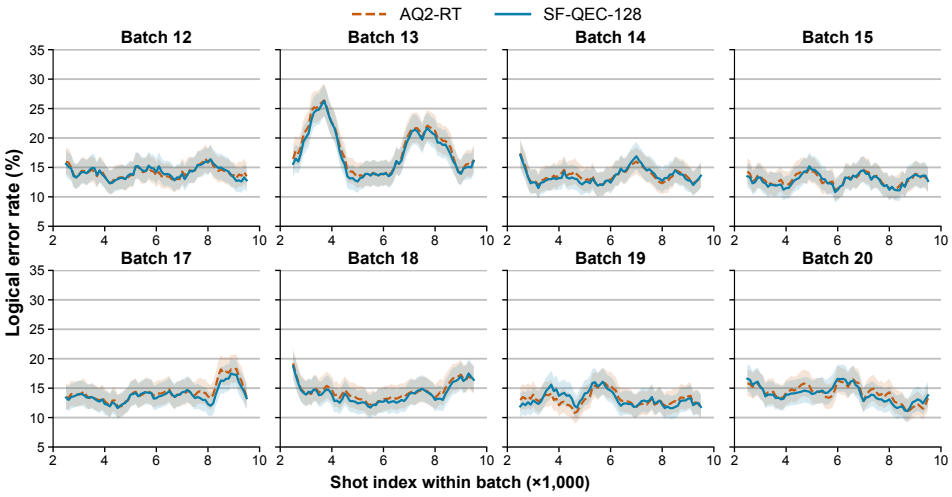


Figure C.16: Rolling-window LER at $r = 6$, patch 3, width 128.

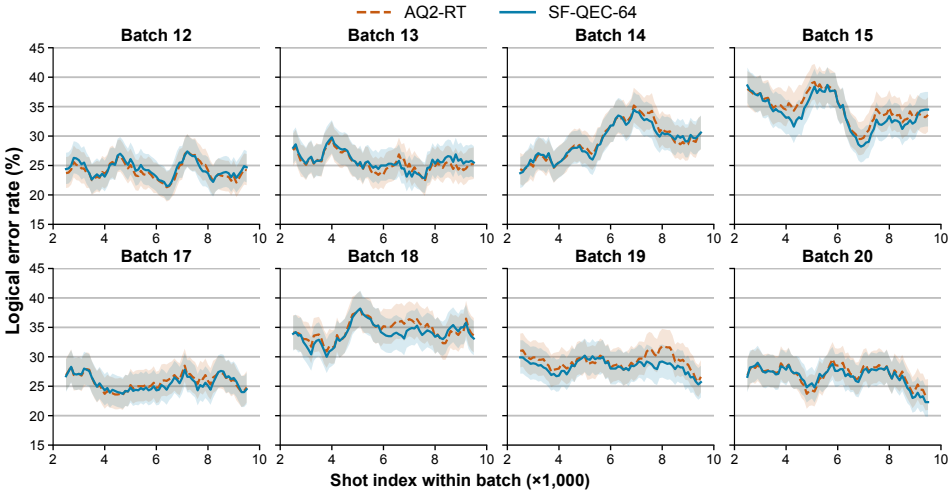


Figure C.17: Rolling-window LER at $r = 9$, patch 0, width 64.

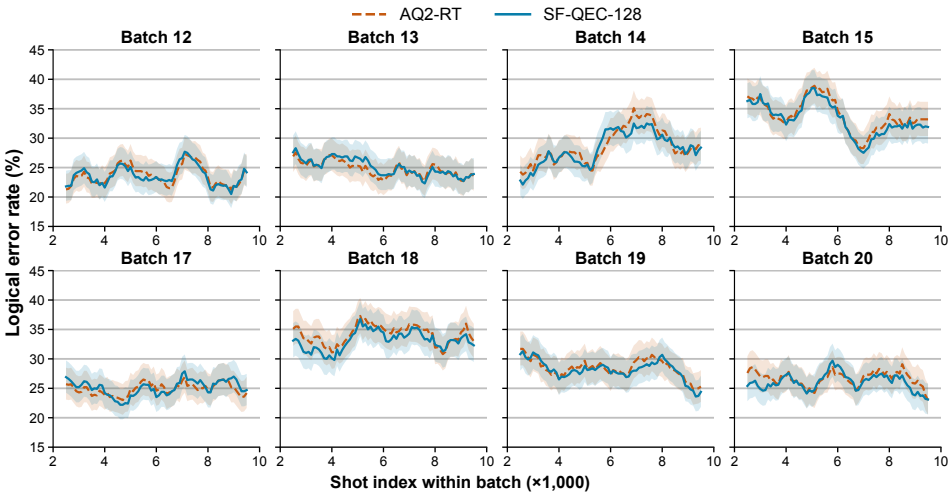


Figure C.18: Rolling-window LER at $r = 9$, patch 0, width 128.

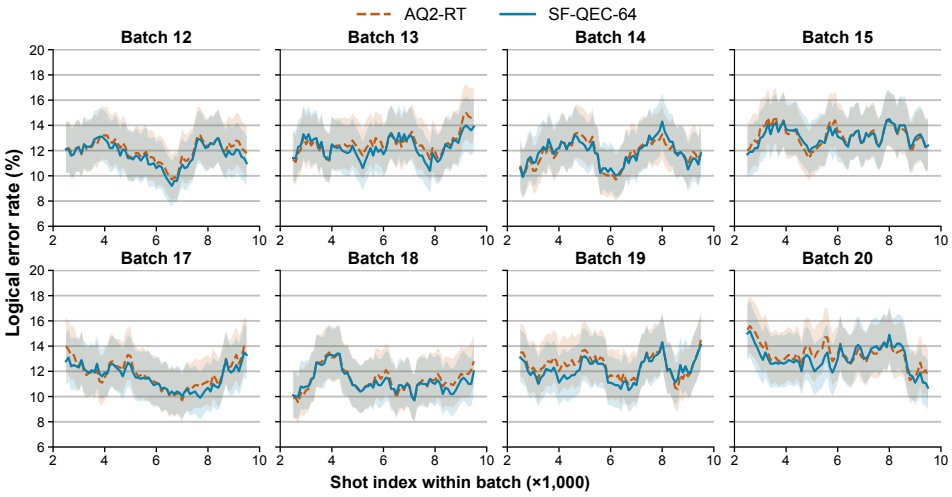


Figure C.19: Rolling-window LER at $r = 9$, patch 1, width 64.

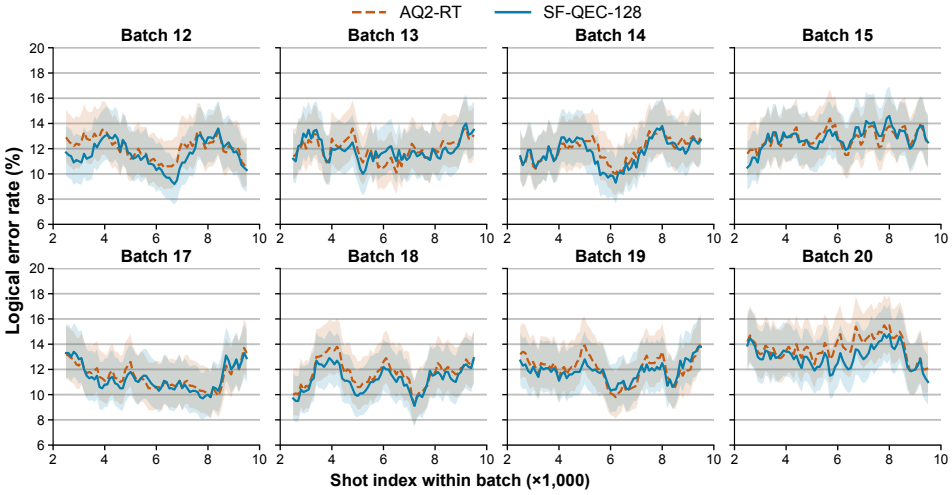


Figure C.20: Rolling-window LER at $r = 9$, patch 1, width 128.

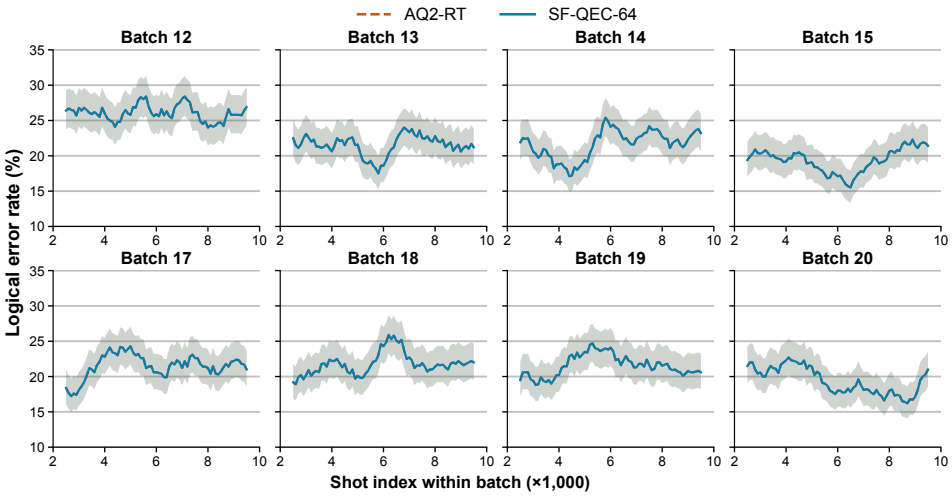


Figure C.21: Rolling-window LER at $r = 9$, patch 2, width 64.

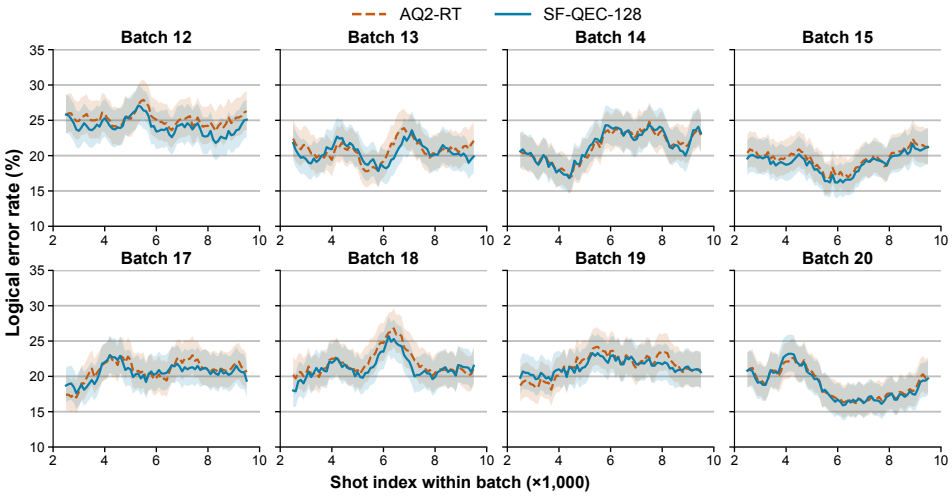


Figure C.22: Rolling-window LER at $r = 9$, patch 2, width 128.

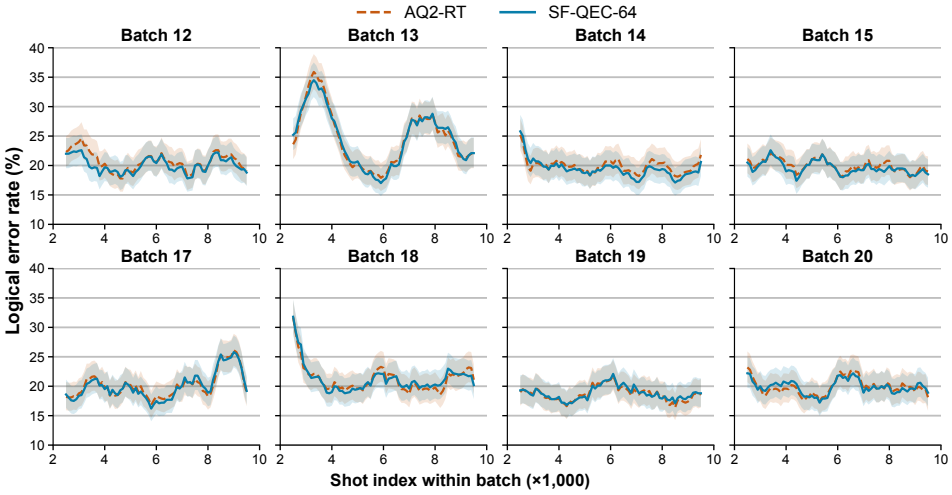


Figure C.23: Rolling-window LER at $r = 9$, patch 3, width 64.

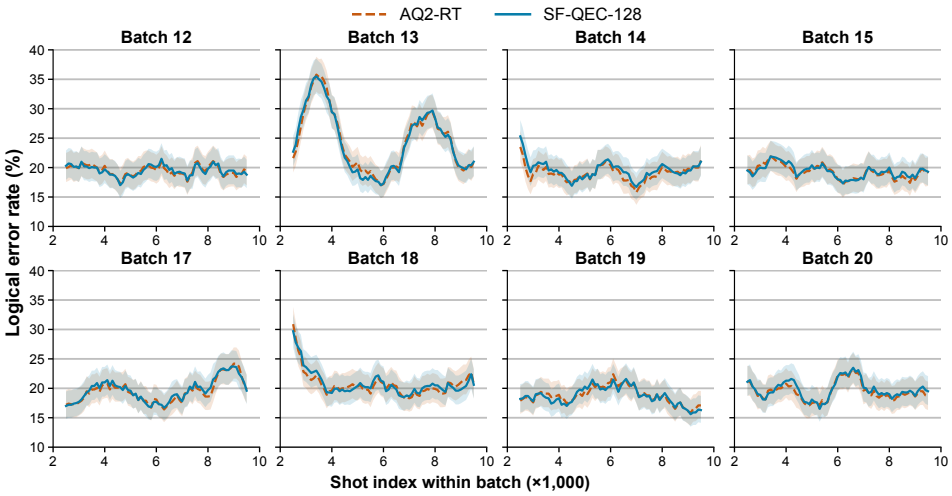


Figure C.24: Rolling-window LER at $r = 9$, patch 3, width 128.

C.4 FP16 ACCURACY

Table C.3 compares FP32 and FP16 using the same trained weights and 64,000 test shots. "Changed" counts predictions that differ between precisions, including both corrected errors and new errors.

Table C.3: FP32 and FP16 results on the same 64,000 test shots.

r	Patch	Widths	FP32 errors	FP16 errors	Changed	FP16 LER (%)
3	0	64/64	6961	6959	4	10.873
3	0	128/128	6865	6865	2	10.727
3	1	64/64	2196	2196	4	3.431
3	1	128/128	2163	2164	3	3.381
3	2	64/64	4128	4129	1	6.452
3	2	128/128	4154	4153	1	6.489
3	3	64/64	4881	4881	2	7.627
3	3	128/128	4903	4903	2	7.661
6	0	64/64	12924	12925	9	20.195
6	0	128/128	12813	12813	2	20.020
6	1	64/64	4970	4972	6	7.769
6	1	128/128	4919	4918	1	7.684
6	2	64/64	9371	9371	4	14.642
6	2	128/128	9120	9119	7	14.248
6	3	64/64	9164	9163	5	14.317
6	3	128/128	9135	9137	6	14.277
9	0	64/64	18211	18216	21	28.463
9	0	128/128	17784	17779	9	27.780
9	1	64/64	7761	7762	7	12.128
9	1	128/128	7638	7637	5	11.933
9	2	64/64	13826	13835	19	21.617
9	2	128/128	13370	13370	6	20.891
9	3	64/64	13210	13214	18	20.647
9	3	128/128	13023	13022	13	20.347

C.5 ADDITIONAL CLASSICAL CONTROLS

Table C.4 compares projected MWPM, DGR with fixed weights, and sliding-window MWPM. The last method updates edge weights using statistics from preceding detector records, without logical labels. All three use our implementations.

Table C.4: Additional classical controls: LER (%) on 64,000 test shots.

r	Patch	Projected MWPM	DGR fixed	Sliding-window
3	0	12.373	12.369	16.337
3	1	4.130	4.230	4.594
3	2	8.869	8.880	8.019
3	3	8.820	8.819	11.584
6	0	22.011	22.061	29.894
6	1	9.459	9.511	10.347
6	2	18.183	18.191	17.838
6	3	16.481	16.481	21.387
9	0	29.666	29.642	38.045
9	1	14.872	14.902	16.783
9	2	25.433	25.447	25.611
9	3	22.344	22.342	27.836

C.6 PERFORMANCE ON SYNTHETIC DATA

Table C.5 evaluates AQ2-RT on 1,000,000 independently sampled shots from the fixed batch-11 DEM for each setting. Both LER columns use this same synthetic test set: "Before" uses the model

after synthetic pretraining, and "After" uses the model after fine-tuning on experimental data. Fine-tuning increases synthetic LER in every listed setting. Epoch counts refer to synthetic pretraining.

Table C.5: Synthetic-test LER (%) before and after experimental fine-tuning. Epochs: selected / completed during synthetic pretraining.

r	Patch	Width	Epochs	Before	After
3	0	32	46 / 50	3.320	3.911
3	0	64	46 / 50	3.304	4.226
3	0	128	31 / 36	3.311	4.091
3	1	64	37 / 42	0.287	0.715
3	1	128	32 / 37	0.286	0.687
3	2	64	47 / 50	0.605	1.108
3	2	128	21 / 26	0.621	1.207
3	3	64	49 / 50	0.851	1.169
3	3	128	27 / 32	0.858	1.158
6	0	32	49 / 50	3.565	4.853
6	0	64	49 / 50	3.538	4.972
6	0	128	49 / 50	3.520	5.135
6	1	64	50 / 50	0.455	1.504
6	1	128	49 / 50	0.449	1.520
6	2	64	35 / 40	0.889	2.208
6	2	128	47 / 50	0.883	2.126
6	3	64	50 / 50	1.170	1.921
6	3	128	50 / 50	1.150	1.922
9	0	32	47 / 50	3.880	5.400
9	0	64	46 / 50	3.806	5.040
9	0	128	46 / 50	3.753	5.201
9	1	64	46 / 50	0.625	2.026
9	1	128	49 / 50	0.592	2.044
9	2	64	49 / 50	1.182	3.041
9	2	128	49 / 50	1.137	3.005
9	3	64	48 / 50	1.530	2.352
9	3	128	48 / 50	1.485	2.312

C.7 Latency measurements

Tables C.6 and C.7 report latency using the protocol in Appendix B.6. "Frozen" uses Inductor compilation with constant-weight precomputation and CUDA graph replay. "Casts" also preserves intermediate FP16 rounding. The median and range are computed over seven trial means.

Table C.6: A100 latency (μs per call).

r	Widths	Path	Precision	Execution	Median	Range
3	64/64	fast	FP32	frozen	391.4	391.4–391.4
3	64/64	slow	FP32	frozen	796.0	795.9–796.1
3	64/64	fast	FP16	frozen	347.3	347.3–347.3
3	64/64	slow	FP16	frozen	669.5	669.5–669.6
3	128/128	fast	FP32	frozen	438.0	437.9–438.0
3	128/128	slow	FP32	frozen	972.8	972.7–972.9
3	128/128	fast	FP16	frozen	366.8	366.8–366.8
3	128/128	slow	FP16	frozen	743.8	743.8–744.0
6	64/64	fast	FP32	frozen	390.9	390.9–390.9
6	64/64	slow	FP32	frozen	843.8	843.6–844.0
6	64/64	fast	FP16	frozen	347.3	347.3–347.3
6	64/64	slow	FP16	frozen	707.7	707.6–707.8
6	128/128	fast	FP32	frozen	434.7	434.7–434.7
6	128/128	slow	FP32	frozen	1037.8	1037.7–1037.9
6	128/128	fast	FP16	frozen	359.5	359.5–359.5
6	128/128	slow	FP16	frozen	794.5	794.4–794.7

r	Widths	Path	Precision	Execution	Median	Range
9	64/64	fast	FP32	frozen	498.9	498.9–498.9
9	64/64	slow	FP32	frozen	871.4	871.2–871.5
9	64/64	fast	FP16	casts	432.3	432.3–432.3
9	64/64	slow	FP16	frozen	747.9	747.9–748.1
9	128/128	fast	FP32	frozen	566.0	566.0–566.0
9	128/128	slow	FP32	frozen	1100.3	1100.0–1100.7
9	128/128	fast	FP16	frozen	459.3	459.3–459.3
9	128/128	slow	FP16	frozen	838.8	838.6–838.9

Table C.7: GH200 latency (μs per call).

r	Widths	Path	Precision	Execution	Median	Range
3	64/64	fast	FP32	frozen	345.6	345.4–345.6
3	64/64	slow	FP32	frozen	706.6	706.4–706.8
3	64/64	fast	FP16	frozen	292.9	291.9–293.6
3	64/64	slow	FP16	frozen	605.5	605.4–605.6
3	128/128	fast	FP32	frozen	1545.7	1545.6–1546.0
3	128/128	slow	FP32	frozen	3386.9	3377.4–3394.8
3	128/128	fast	FP16	frozen	306.0	306.0–306.0
3	128/128	slow	FP16	frozen	658.1	656.4–658.2
6	64/64	fast	FP32	frozen	344.4	344.4–344.6
6	64/64	slow	FP32	frozen	733.1	732.8–733.9
6	64/64	fast	FP16	frozen	296.7	296.7–296.7
6	64/64	slow	FP16	frozen	626.5	626.4–626.5
6	128/128	fast	FP32	frozen	1552.2	1551.9–1552.6
6	128/128	slow	FP32	frozen	3422.1	3413.8–3429.0
6	128/128	fast	FP16	frozen	307.1	307.1–307.1
6	128/128	slow	FP16	frozen	652.4	652.3–653.7
9	64/64	fast	FP32	frozen	435.1	435.1–435.4
9	64/64	slow	FP32	frozen	793.9	793.2–794.8
9	64/64	fast	FP16	casts	366.2	365.3–366.7
9	64/64	slow	FP16	frozen	626.3	625.4–627.3
9	128/128	fast	FP32	frozen	2102.7	2102.1–2103.0
9	128/128	slow	FP32	frozen	3465.2	3450.7–3471.4
9	128/128	fast	FP16	frozen	388.4	388.4–388.4
9	128/128	slow	FP16	frozen	704.4	703.5–704.9

The width-128 FP32 runs are slower on GH200 than on A100 with the tested software environments. Repeating the $r = 3$ case in a new process gives 1558.1 μs for the fast path and 3414.0 μs for the slow path, consistent with the original measurements.